RAY TRACING AS THE FUTURE OF COMPUTER GAMES

Juri A. Oudshoorn

Department of Computer Science, University of Utrecht
In conjunction with Davilex Software
November 1999

Abstract

To increase graphical realism in future 3D computer games, ray tracing might become a rendering technique used. This, because ray tracing is easy to implement and it can deliver very realistic graphics. Unfortunately, ray tracing is very computational intensive and as such not suited for real-time application... yet.
Current computer games add various tricks to real-time rendering to increase their level of realism and still remain interactive framerates. To find out if ray tracing will ever substitute this, several advanced real-time techniques are reviewed in this thesis. Techniques relating to objects, namely curved surfaces, levels of detail and bump mapping, are reviewed first. Secondly, techniques relating to light, namely reflection, refraction, shadows and indirect lighting, are reviewed. Animation techniques, namely skeletal deformation and weighted vertices, and blurring techniques, namely antialiasing, motion blur and depth of field, are also reviewed.
Each of the techniques is first briefly explained. Then, for each of the techniques several methods are named that can achieve the desired effect and for each of them, the thesis details advantages and problems as relating to real-time rendering. Finally, if existing, several examples are named of games in development that utilise these methods and thus is estimated if the techniques will be used in future computer games. This way a comparison can be made between the benefits and drawback of ray tracing and traditional real-time rendering.

Contents

Chapter 1 - Introduction

Will ray tracing be the future of computer games? This is an interesting question indeed. Some people believe that real-time ray tracing will become possible, while others dismiss it as science fiction. This paper will try to answer the question by looking at several advancements in the technology of computer game graphics. These techniques will be related to offline rendering methods as to answer the questions: “Which techniques from ray tracing will be incorporated in computer games?” and “Which techniques will computer games incorporate from other rendering methods?” Answering these questions will shed light on the question: “What will be the future of computer games?” This question in turn will enable the answering of the question if ray tracing will be the future of computer games.

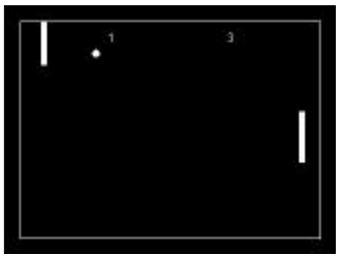
Chapter 1 will introduce *computer games*, *realism*, and *ray tracing*. Chapter 2 will look at where 3D computer games graphics currently stand. Chapter 3 will look at future techniques that have to do with objects. This includes a look at *curved surfaces*, *levels of detail*, and *displacement mapping* (including *bump mapping*). Chapter 4 will look at future techniques that have to do with light. This includes a look at *reflection and refraction*, *shadows*, and *indirect lighting* (specifically *radiosity*). Chapter 5 will look at future techniques that have to do with animation, specifically *bone animation*, *weighted vertices*, and *physics*, and that have to do with blurring, specifically *antialiasing*, *depth of field*, and *motion blur*. Chapter 6, the concluding chapter, will look at the future of 3D computer game graphics taking the information from the previous chapters.

1.1 Computer Games

What are computer games? While, at first this question seems simple to answer – computer games are games on computers – this does not give a whole lot of information. It foregoes to mention that games are a combination of stories, art, music, sound effects, animation, and programming techniques in a dynamic, interactive form of electronic entertainment ([76]). They exist on various platforms, PCs, consoles (video games), and networks (multiplayer games), although the latter is, due to its nature, usually paired with one of the two former. This leaves the question as to what kinds of computer games there are.

There are as many different games as there are people making them. Some are more original than others. To place them into categories it makes sense to look at *viewpoint*, *dimension*, *genre*, *play type*, and *action resolve*. The different viewpoints of games are the same as in books: first person, where the player sees through the eyes of the character he’s controlling, and third person, where there player looks at his character from the outside. Dimension refers to the issue of 2D versus 3D. Play type divides into single player, where the player plays alone, usually through pre-set events, and multiplayer, where the player plays against other players. In addition, action resolve refers to turn-based, where each action is taken in turn, versus real-time, where the clock keeps going whether you do something or not.

The distinction in genres is somewhat more complex, especially since the different genres are not as clearly defined as in maybe the film-industry and because many games seem to fit into several groups. Yet, [76] made a very good attempt, which has been adopted here, slightly edited: Adventure games and interactive fiction, edutainment games, fighting games, god games, platform games, puzzle and card games, role-playing games, shooter games, simulation games, sport games and strategy games. For examples in each genre, see Appendix 2 on page 46. Where did all these games come from?



Figuur 1 - Pong

It all started in the early 1960s with programmers in various labs developing games. The arcade game Pong became a major hit after its introduction in 1974. The company that had developed it, Atari, started moving its products into homes and introduces a cartridge system. Companies like Nintendo (with Donkey Kong) and Namco (with Pac-man) took over the arcade industry. Then in 1982 the market crashed, the industry dropped from over $6 billion to below $800 million, and many companies went down with it.

Personal computers, which have been around since the early 70s, started taking off, first with computers like the Apple II, Commodore 64 and Atari 800. Later computers like the Atari ST, Commodore Amiga, Apple Macintosh and IBM clones started setting a new pace. Nintendo resurfaced in 1984 and recreated the home video market. It found rivals in Sega Enterprises in 1986 and Sony in 1995. Atari ST and Amiga failed and the MS-DOS platform took off. In the 1990s, PCs with sharp graphics, professional-quality sound capabilities, and CD-ROMs started to become commonplace.

Figuur 2 - Doom

With Doom in 1994, 3D graphics broke through on PCs, which resulted in a whole plethora of clones. In 1995, Microsoft's Windows 95 and the Windows Game SDK (now DirectX) were released and Internet started to grow. The follow up to Doom, Quake, was released, giving birth to widespread multiplayer gaming, in 1996. A war started between 3D acceleration cards, with 3Dfx as the temporary winner. The market saw a surge of 3D games, some pushing the envelope even further. Recently announcements of new consoles boast 3D graphics superior to those of PCs, which is quickly battled by the announcement of a new 3D card for the PC. Online gaming grows ever stronger and the distinctions between genres start to blur.

So why have computer games been chosen as the research area for this paper? There are several reasons for this. Most importantly, computer games technology is a very fast evolving field. So much so that at times the only reason to buy a new computer seems to be able to play the latest computer games. Computer games have a tendency to include the latest state-of-the-art graphical technologies, within the real-time constraint described in the next paragraph. This makes computer games very suitable to research technological advancements in real-time 3D graphics and opens up the question as to what consumers might expect in their future games.

As has already been stated, one of the constraints 3D computer games have is that they generally have to be rendered real-time. While there are 3D games that use pre-rendered 3D graphics, they really do not differ much from hand-drawn scanned art. While, due to the speed limitations of present day computers, some effects might not be realisable in real-time, the benefit of real-time is that it draws the observer in the virtual world. This way realism in a game can be judged by the degree of the observer's immersion in the world.

Furthermore, computer games are big business. In 1994, the entire computer games industry pulled in over $6 billion worldwide ([76]). It only increased from there. U.S. video game retail sales alone equalled that amount in 1998 ([60]). Online gaming is not only rising, from around 7 million players in 1998 to a projected 18 million in 2001, but it is also becoming a profession. Players can challenge each other in professional competition leagues, with prices in the tens of thousands ([30]). Also programming for (or generally developing) games is considered increasingly as serious work ([82]).

Another consideration is that the author of this paper works for Davilex software, a Dutch company that creates computer games. The game that the author helps create can function as a running example. Since it is a state-of-the-art first person shooter, it serves perfectly well in current technology. It also serves as an example of problems with current technology and it can suggest how they could be fixed. Finally, and certainly important, computer games are fun. While it might be possible to do something similar with architectural walkthroughs, that would not be very entertaining.

Throughout this paper, various computer games will be referenced as example. Note that they all fall in the first person shooter genre. There are reasons for this. First, the first person perspective is closest to how people daily experience the world. Because this view is more familiar, it is easier to compare the realism of the graphics with the real world. Second, first person shooters tend to be on the cutting edge of real-time 3D graphics. Since this paper mostly looks towards the future, this is important. However, one game serves as a running example throughout the paper.

Amsterdoom by *Davilex Software* is based on a real city, namely Amsterdam (the capital city of the Netherlands). It is interesting for various reasons. First, because it is based on a real city, it can serve as a good comparison for purposes of realism. Second, it serves as an example of current technology standings and shortcomings in it. Finally, the author of this paper is involved with the development of this game and as such can use it to shed light on the inner workings of certain aspects of 3D games. In Amsterdoom, aliens have taken over Amsterdam and the player has to kick them out. Doing this the player will visit locales such as the central station, the main street, the airport and the red light district of Amsterdam. The game is set for a release in March 2000 in the Netherlands. Of course, it will not be the only example in this paper.

Doom started the first person shooter genre in 1994 and it was soon followed by *Doom 2*. Yet, Doom was not true 3D (many debates have been waged over this, finally compromising on the term 2½D). Therefore, when its follow-up *Quake* was announced, the world of computer games entered a new era. It was the first true 3D first person game and today it almost seems as if the only way to make a game is in 3D. It was followed by *Quake 2* and now *Quake 3 – Arena* is near completion, focussing solely on multiplayer gaming. Due to Quake's high customisability, it spawned user created modifications such as "capture the flag" and "team fortress". The latter one was so popular that now a stand-alone game *Team Fortress 2* is being created.

Figure 1 - Wheel of Time

The major competitor to (especially) Quake 2 was the surprise hit *Unreal*. While many have claimed that its gameplay is below Quake's, none could rightfully claim that its graphics were less, since they were beautiful. With advanced effects like its amazing lighting effects, fractal textures, and volumetric fog ([49]), it looked more realistic than any game before. Now *Unreal Tournament* is set to compete with Quake 3 and the power of the Unreal engine has more in store. It is being used in the very ambitious title *The Wheel of Time* (WoT), which is based on the best-selling fantasy series by the same name. It looks to be even more realistic and amazing than Unreal, due to its solid architectural level design, high-fantasy art style texturing, and incredible set of graphical features added to the Unreal code base. Moreover, the world of computer games may have more surprises in store for us.

1.2 Realism.

Realism is a highly problematic concept with different connotations in different contexts. Popularly it refers to scepticism about what can be achieved through action or sometimes to a lack of ideals or principles ([74], [10]). Philosophy gives two more senses. In scholastic philosophy, realism means that universals have a genuine, tangible existence ([41], [74], [7]). Metaphysical realism however states that real objects exist independently of them being observed ([20], [74], [7], [10]). Yet, the most useful definition seems to come from art and literature. Here realism is described as the realistic and natural representation of people, places and/or things in a work of art, usually further explained by an absence of idealisation ([15], [80], [3], [7], [10], [42]).

This definition is somewhat problematic, besides the fact that pure realism, the idea of copying reality directly, is a myth. This is because every depiction is always an abstraction to some extent (which will not be discussed further in this paper, see [42] for more on this). There is also the fact that most computer games do not even try to copy reality. Computer games try to be entertaining, involving, beautiful or sometimes just marketable, but rarely do they try to copy reality. In fact, they are often used as a means to escape reality. Even Amsterdoom (see 1.1) which takes place in some existing locations often bends the rules (if a location is problematic, just shut a part off, or change the geometry a little, or include some alien technology).

So of what use is the definition? Maybe it is better to start with the term *realistic*. Since, while games do not aim for realism, they can often be realistic. Realistic can mean two things ([85], [10]): either to accept things as they are and dealing with it practically, or appearing to be existing or happening in fact. While the first meaning is not very interesting within the realm of computer graphics, the second is. It implies that things that have little or no basis on reality can convey the illusion of being real. In a sense, they create their own realities, each with more or less reference to the real world.

This makes it seem appropriate to make a distinction between two different kinds of realism. First, there is "real-world realism". This is the kind of realism referred to by the definition from art and literature. Nevertheless, as already stated, this kind of realism is not very useful in the context of computer games. The second kind of realism can be called "fantastic realism". This is a temporal realism created by a mix of various existing, changed and non-existing elements. It is a strange kind of realism, since everybody knows that it is not real. Yet, somehow it has a certain degree of believability.

Closely tied with this distinction is the *suspension of disbelief*. To suspend disbelief is to believe in something that you know is not real, or to act as if ([10]). By creating a fantastic reality, one allows an observer to pretend that it is real, to believe in this fantastic reality. While watching a movie, for instance, one can be completely drawn into the world created by the filmmakers. The role of realistic graphics in this is to increase the observer's ability to suspend disbelief by lessening the amount the observer has to work to believe it ([25]).

Thus the definition of realism can be stated as follows: the realistic representation of a reality, whether a real-world reality or a fantastic reality. Now this definition of realism can be used as a goal for computer games. Nevertheless, it leaves the question open as to why realism is a goal of computer games. While realism can lead to facilitate the suspension of disbelief, that does not really give an answer.

To understand why realism is important, it is first important to understand what *immersion* is. According to [47], [85], [11] and [65] immersion is (within context) the state of deep engagedness, being overwhelmed, being deeply absorbed or being completely involved. Meaning, in the case of real-time graphics, that a scene is so mesmerising that, at least temporarily, an observer can accept it as a reality.

This does not mean that the image has to be realistic. According to [25] immersion has the effect of allowing your brain, almost effortlessly, to fill in graphical gaps in realism. This way one can be immersed in a book (as an extreme example) by filling in the graphical gaps in realism from the description giving in the text

(as well as imagination and memory). Thus immersion creates or increases a sense of realism in the observer's mind, although he is very aware that what he is seeing (in his head or on a screen) is not real. Or, more correctly, it facilitates the suspension of disbelieve.

So why is immersion so important? It is not the first thing that comes to mind when thinking about the goal of computer games. Games aim at creating an entertaining experience. In this, immersion is more of a by-product of several factors coming together in the "right" way and entertainment is certainly one of those factors. Thus, immersion is a measurement as to how "right" everything works together. This makes immersion one of the ultimate goals of computer games. Since if all the separate elements (gameplay, graphics, sound, story, etc) work well and (more importantly) they work well together, then the game or art attains immersion.

However, it is very easy, indeed quite common, for one person to be easily immersed in something while another simply can not be immersed in it easily. After all, immersion is still very much the state of someone's involvement. Since everybody is different, that "right" is defined differently by everyone. Where for one person not beauty of graphics but speed of play is more important, another might be turned away by fast but ugly visuals. Thus, immersion is one of the most illusive goals for a game to aim at (together with gameplay, which also differs from one person to the next).

So, if immersion is the ultimate goal of computer games, why do we want realism? As stated above, immersion creates or increases a sense of realism. However, it works both ways. If something looks believable, it is easier to be immersed in it. In addition, if something looks realistic it is more believable. Of course, the question remains open if something unrealistic can be believable. As per the real-world realism versus fantastic realism, unrealistic things create their own form of reality. At least realism affects immersion and since immersion is the ultimate goal of computer games, the importance of realism has been shown.

Now that it is clear what realism is and why it is important, the question rises: what influences realism? While realism consists of many factors – sound, graphics, environment, interface, playability, AI, etc – this paper focuses on graphics and so will this section. Leaving us with the question: what influences graphical realism? For this, four areas can be explored, namely: *geometry*, *surfaces*, *lighting* and *animation*.

Geometry is everything that makes up the world, particularly walls, floors, etc. Besides largely defining the environment, geometry also affects realism by its representation and limitations. For instance, realism might be affected by the restriction of geometry not being able to used *curved surfaces* and *level of detail* is a way of handling geometry to speed up calculations and might influence realism. Therefore, it is important to look at various aspects of geometry regarding realism.

Where geometry makes up the world, surfaces dress it up. By applying images to the surfaces defined by the geometry, one creates the illusion of materials. Because these images are 2D, where surfaces in the real world rarely are, it is important to look at things like *displacement mapping*. In addition, how surfaces react to light is important, so there needs to be looked at *shading* and *radiosity*. One must conclude that surfaces are important to realism as well.

Lighting is often named as the most important influence to realism (see for example [25], [81]). This means that one cannot look at realism without looking at lighting. Specifically things like reflection, transparency and refraction are important, as do (dynamic) light and shadows and various blurry effects like anti-aliasing, soft shadows and distance fields.

Finally, after lighting, animation is usually the second thing noted as very important to realism. Especially in a real-time environment like computer games, animation is often a deciding factor in realism. Issues like skeletal animation, vertex manipulation, and physics simulation can influence how realistic something seems. Therefore, these things have to be looked closer at.

Computer games already use a variety of techniques to improve realism. While this will be handled in more detail in Chapter 2, a quick overview will be given here. Due to the real-time nature of computer games, they have to use various tricks to achieve realistic images and still have interactive framerates. One of these tricks is to pre-calculate a lot. This includes geometry, lighting and visibility. Because of this, high quality lighting calculations can be done at virtually no cost to the eventual game (depending on the resolution of the light map, see section 2.2 for more).

Having pre-calculated light adds a lot of realism to real-time environments in the sense of a single image being realistic. However, when a player moves though such an image, one can feel the unchanging nature of the world. This is where dynamic lights come in. While these lights are more expensive in real-time calculations and often of less quality, they add a certain dynamic to light in general, conveying the feeling that light can change at any time. There are generally two ways to have dynamic lights, either add extra lights real-time, or influence the pre-calculated values.

Tied closely together with light are shadows. Since they depend on light, they are often pre-calculated together with lights. However, with dynamic lights and dynamic objects, shadows have to be calculated real-time. While leaving them away can look unrealistic, adding them can be a quite demanding calculation. This is

why usually tricks are used, like placing a sprite of some blurry spot on the ground below the object, representing a shadow. However, real-time shadow calculations are increasingly used.

Because computer games offer a real-time environment, animation is also very important to realism. On way computer games get realistic animations is by capturing the motions of real-world objects and applying them to the appropriate objects in the game. This is usually done by pre-calculating each frame in the animation sequence and just playing through these frames during the game. Another way to do it, a way that is becoming increasingly popular, is by animating a skeletal structure of the object and calculating the animation of the object in real-time by this skeleton. This allows for greater flexibility in the animation, for instance because of being able to combine various animations.

In geometry, the way to increase realism is usually by keeping the areas small. This way, less calculating time is needed which can be used to add detail. As processor power increases, the areas can become larger and the details more abundant. Another way to increase realism through geometry is in the way it is used. By letting architects design the areas for example, the feeling that the area might actually exist is increased. Another way to do that is by giving the areas a consistent feeling. This can be realised by using a general colour palette, a similar style in texturing, or other artistic tricks like it.

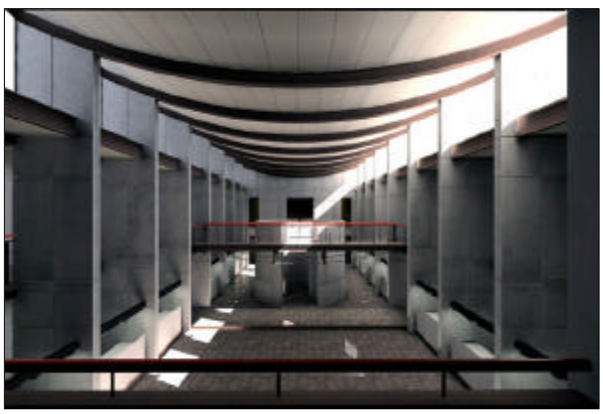
Finally, computer games use a range of special effects to increase realism. These special effects are only applicable in certain locations and much of their benefit is often under discussion. One of the most discussed ones are lens flares, since they look good but one generally does not look through a camera to see the real world. Other things usually considered being effects are fog, particles, and reflection. While this last one is considered an effect, reflections (and to a lesser extent transparency and refraction) are a large part of the real world. Currently, games have certain surfaces marked are reflective which gives it some mirror-like properties, but the general blurry reflectiveness of real-world surfaces must either be mimicked in textures or is not done at all.

Of course, the final consideration is that games have to be entertaining and realism often has to give way to increase the fun of a product. Another consideration is that games have to make money, so certain aspects of realism might not be implemented because of deadline, funding, or other marketing issues. Luckily, real-time 3D technology is evolving at a high rate; so more and more becomes possible to achieve realism.

1.3 Ray Tracing

Ray tracing is a method to determine the visibility of surfaces within a 3D object-space by tracing imaginary rays of light from the camera to the objects in the scene ([23]). A ray is traced from the camera through each pixel of the screen, which is represented by a window on a view plane. The first object this ray intersects with is the closest object and thus the visible surface is part of this object. The colour of the pixel is then set to the colour of the object at the intersection.

The above method is also called *ray casting*. The term ray tracing is usually used for *recursive ray tracing* in which, upon intersection, more rays are generated. The benefit of this is that it can generate realistic shadows, reflection, and refraction. To calculate shadows, *shadow rays* are cast from the intersection point to each of the lights. If one of these rays intersects any object, then the contribution of this light source is ignored, since it is in the shadow of the object. *Reflection rays* and *refraction rays* are cast if the object is reflective or transparent respectively. Reflection rays are reflected about the surface normal and refraction rays are sent into the object at an angle determined by the refraction coefficient object. Both reflection and refraction rays may recursively spawn reflection, refraction, and shadow rays until they don't intersect any objects, or some user-defined maximum depth is reached. The results of the rays are added together according to some coefficient to produce the final pixel colour.



Figuur 3 - Image made with Lightscape

While this process creates stunningly realistic images, it is also very slow. Recursively calculating each of those rays takes an enormous amount of processing time. Not to mention shooting multiple rays through one pixel to generate anti-aliasing (see 5.2), which increases the number of calculations dramatically. Rendering a single image can take anywhere from a few seconds to several days, depending on the scene complexity. Thus, it does not seem suited for real-time environments.

Of course, ray tracing is not the only method for creating photo-realistic images. Radiosity, for example, calculates the light interactions in an environment in a view-independent way. Surfaces are divided in

patches, each with their own light value. These patches emit light to the other patches and receive light back from them. This process can be repeated in multiple passes to generate better effect. Then views can be rendered, with only the overhead of visible-surface determination and interpolative shading.

Then why use ray tracing? Most importantly, ray tracing produces high-quality photo-realistic images. Light is one of the most important things in creating realistic pictures ([25], [81]) and ray tracing can simulate light very well, especially reflection, transparency, refraction, and shadows. Other things, like texturing, it can do as well or better than other rendering methods. Current approaches cannot do true shadows, reflection, and refraction, only a reasonable simulation of them.

Being very computational intensive is not the only problem with ray tracing. The method is also not very good at global illumination; at least, a better method exists. This is where the speciality of radiosity lies; it is very good at diffuse reflection between coloured surfaces. Radiosity has its problems too though. Sharp changes in colour can be a problem, especially with shadows ([69]). It also cannot render reflections on shiny surfaces and it cannot handle transparency and refraction very well. The best solution seems to use a combination of radiosity and ray tracing, but that does not eliminate the high calculation time needed for ray tracing. See 4.3 for more information on global illumination.

With the problems associated with each method and the benefits each has, it is impossible to conclude at this point that ray tracing is the only thing to look at. Ray tracing offers an intuitive, but costly, way to simulate rays of light. Radiosity offers very realistic indirect lighting, but at the loss of various other effects and high pre-processing requirements. Traditional methods are faster, but require the use of tricks to simulate the effects ray tracing offers naturally and they do not have the quality of global illumination of radiosity. Therefore, it is important to keep an eye on all methods, but with a specific eye on ray tracing, since that method seems to create the most realistic images.

Three-dimensional computer games currently use a polygonal, scan line based rendering algorithm ([76]). They have taken techniques from ray tracers, however. The BSP-tree (see 2.1) for instance was originally created for ray tracing. Thus, the consensus is that current ray tracing techniques might also be incorporated in 3D computer game technology. Specifically thing like shadows, reflection, and refraction, where the strength of ray tracing lies, must be examined for games. In addition, issues like indirect lighting, shading and curved surfaces must be examined.

With the scan line methods game currently use, various tricks must be applied to simulate the desired effects. Since these calculations can be very intensive, they are often used only sparsely or not at all. Shadows are pre-calculated for static objects and are rarely implemented for dynamic objects (although dynamic light often can cast shadows from static objects, due to being pre-calculated as well). Reflections are used sparsely and only in very controlled and small areas, so that there is never too much reflected. Reflection is also used exclusively for mirrors and very shiny surfaces. Transparency alone is hard, and rarely correctly enough implemented, let alone refraction, which is never used in computer games so far.

Using ray tracing in computer games could really boost realism. Ray tracing is still very expensive though and computer games need fast algorithms over realistic algorithms. So this paper must research two things: “When will ray tracing be possible for games?” and “What aspects of ray tracing can games incorporate?” These questions will be answered in Chapter 6.

Chapter 2 – The Present

To understand how the future of computer game graphics looks, it is first important to understand how the current situation is. This chapter discusses the most common terminology and techniques as currently used. It starts with discussing the *geometry* that makes up the worlds in section 2.1 below. Then, in section 2.2, it discusses how various material properties can be applied to surfaces and what extra considerations there are in that. In section 2.3, lighting and shading are described, as well as how they are realised in computer games. Finally, section 2.4 is about animation and special effects.

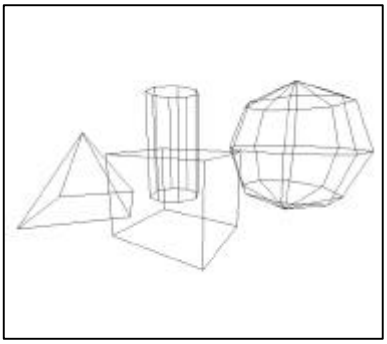


Figure 2 - Brushes

2.1 Geometry

The worlds of computer games are built up of various objects. Namely: *brushes*, *models*, *actors*, and *entities*. It is important to understand the

differences between these building blocks. Amsterdoom, like almost every first person shooter, has all of these things.

Brushes are used to make the static parts of the world, walls, floors, ceilings, stairs, columns, etc. These 3D shapes can be cubes, cylinders, pyramids, spheres, etc. Currently the sides (surfaces) of the brushes have to be (usually three- or four-sided) flat polygons, so brushes are a subset of polyhedrons. Brushes also usually have to be convex, although some editors allow concave brushes. There are various types of brushes. Solid brushes are the most common; they are visible and cannot be moved through. Clipping brushes are invisible, but cannot be moved through. Empty brushes are visible and can be moved through. They are usually used to create liquids, although some editors have a special brush type for liquids. Hint brushes are invisible and can be moved through. The sole purpose of this brush is to help the compiler. Sheet brushes are brushes consisting of a single surface.

So there can be subtraction brushes, which are subtracted from the world instead of added, and are used to make holes. Some editors implement this by replacing a brush by several brushed that consider the hole. This is just a trick compared to true constructive solid geometry (see below). Amsterdoom uses four-sided surfaces and allows a few concave brushes, but the set of brushes to choose from is very limited (box, sphere, cylinder, stairs, arch, and cone, all of which can be hollow).

Models are brushes with one important difference; they can move. They are used for things like doors, elevators, and platforms. Models have several keyframes, each frame with a different position of the model. During the game, the model animates from one keyframe to another. Due to this, models have very limited animations; they cannot change shape during the animation. Models are useful as area brushes, in which they can seal of a part of the world. Usually doors are area brushes. Due to them being dynamic, models usually do not cast shadows and they often do not respond to differences in light conditions from one frame to the next. There are exceptions (like Unreal) where models are capable of the latter. Amsterdoom however does not have this capability.

Entities are the things you place in a solid world. Some entities, like lights and the Sun entity in Amsterdoom, are used in visualising the world. Others, like the items and enemies of Amsterdoom, have a very tangible appearance in the world. Some, like the Trigger entity and the DoorModel entity in Amsterdoom, take care of the dynamic behaviour, linking the world to the dynamics of the game. Finally, some entities, like PlayerStart and again Triggers in Amsterdoom, are used to recognise certain locations in the world. This might result in dynamics (as is the case with Trigger), or they are just used to have a location reference (like PlayerStart and again Sun in Amsterdoom). Of all the entities, the various types of light are the most important for creating realism. In editors, entities appear as a small icon (an ‘X’ or a small picture).

Actors are closely tied with entities, specifically those with a physical appearance in the game, like items and enemies. They are 3D figures, sometimes with their own animation frames, which are usually made in a 3D graphics package like 3D Studio Max. They can move within the world and are usually far more detailed than models. They can have complex animations, including changes of shape. Actors often have a one-on-one relationship with entities, a single health item actor appears where the ItemHealth entity is placed in the world, but this does not have to be the case. Factory entities create various items or enemies during the game and shooting a rocket usually creates a rocket actor, never having used an entity. Actors are the visible representation of what the player interacts with most in the world (next to the world itself and the interface). When there is a one-on-one relation, some editors show a wireframe of the actor instead of the icon for the entity. Note that the term “model” is sometimes used instead of the term “actor”.

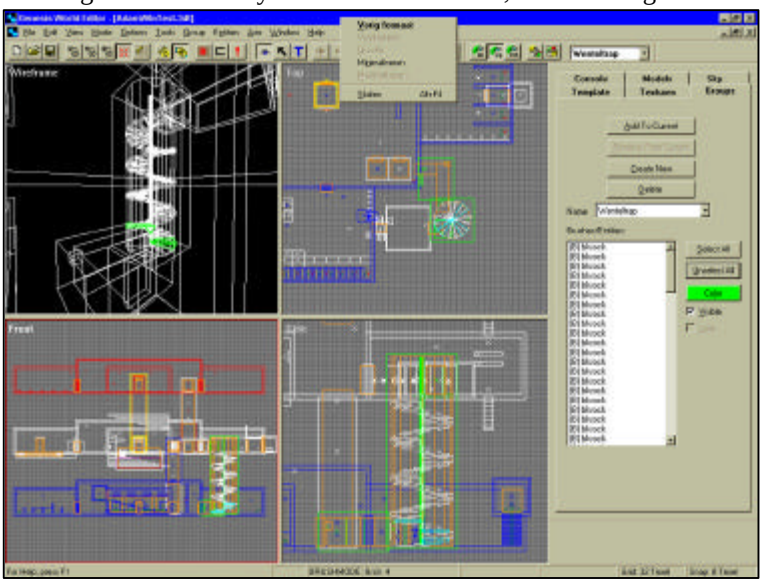


Figure 3 - Amsterdoom editor

Usually, when building a world, brushes are added to the world. In special instances, brushes can be subtracted from the world. Entities are also added to the world and models are created from brushes. One can modify the properties of the brushes, models and entities (make a brush empty, set

the animation frames of the model, and define the strength of the light entity). Actors are either created during the game or linked to an entity; actors are never placed directly.

Some editors, like the Unreal editor ([49]) and partly the Amsterdoom editor, use *Constructive Solid Geometry* (CSG). In CSG, the brushes are combined by means of Boolean operators ([23]). Without CSG, one starts with a void and brushes can only be added to this (as describes above, subtraction is only a trick in these cases). With CSG, one can start with a giant brush and the rooms are carved out from this. In Amsterdoom, while one starts with void, one can do true CSG subtraction. In the end though, they are all converted to a BSP tree (see below).

Once a world is built, it (usually) goes through three stages: converting to BSP, visibility calculations, and lighting calculations. Lighting is handled in section 2.2 and visibility is just a set of calculations that optimise the BSP tree for visibility determination. This includes calculations that determine which surfaces are never visible from certain locations.

A world is converted to a *Binary Space Partitioning tree* (BSP tree) because this data structure is perfectly suited for hidden surface removal. A BSP tree works as follows ([23], [53], [8]): The space is divided in two subspaces by a plane of arbitrary orientation and position. This plane is stored in the root node, everything behind it is stored in the left branch, and everything in front of it is stored in the right branch of the tree. Then the two sub-spaces are recursively divided further into subspaces, until each sub-tree (leaf) only contains one segment of the world. The advantage of a BSP tree is that everything on the same side of a plane as the viewpoint cannot be obscured by anything on the other side of the plane, and can thus be drawn last (in a painter's algorithm).

A refinement to this process can be found in *portal culling* ([53]). The scene is divided into *cells* that usually correspond to rooms and hallways. The doors and windows that connect these cells are called *portals*. With each cell, information on the adjacent cells and the portals that connect them is stored. The cell in which the viewpoint is located is normally rendered. Other cells are only rendered if a straight line (sight line) can be found from the viewpoint (in the viewing direction) to the cell without intersecting geometry. Where a game such as Unreal uses invisible sheet-brushes as area portals, Amsterdoom uses models (specifically doors) as area portal. This means that a cell (area) is not drawn if the door is closed and is drawn if the door is open.

BSP trees are not the only way to divide a space. Another possibility is the *octree*, a higher dimension version of the 2D *quadtree* ([23], [53], [8]). Quadtrees divide an area in both dimensions to form quadrants. Each quadrant is then further divided until each section contains only a single piece, which can often go down to the pixel level. Octrees, instead of quadrants, have octants and are suited for 3D representation. Yet, very few computer games use octrees, they all use BSP trees instead.

Of course, the problem with these techniques is that they are only suited for static environments. Models and actors are a way to get some dynamics back into the environment, but are too costly to use to build the entire world. If one wants to change the geometry real-time, one either has to find a better implementation, find out how to change the BSP tree real-time (and preserving the pre-calculated quality), or wait for a faster system. An interesting note is that Quake 3 was originally intended to support real-time geometry changes, for which it wanted to forgo the BSP tree ([57]), but it did not succeed and thus Quake 3 does use BSP files now.

2.2 Materials

A world could be made up of anonymous, mono-coloured surfaces, but that is not very exiting. It is better to be able to indicate for brushes, models, and actors of what material they are made. For this, *textures* can be applied to these objects.

To build a world that looks anywhere near as complex as the real world, one needs thousands of brushes. The reason is that hardly any surface in the real world is completely flat and coloured in a single colour. To avoid this, Amsterdoom and games like it use *texture mapping*. With this technique, a picture is applied to a surface to create the illusion of detail on the surface (also called *image texturing* ([53])). Actually, there are two kinds of texture mapping ([9]). *Texturing* is usually used to refer to the process of applying an image to a brush or model. *Skinning* is used to refer to the process of applying an image to an actor. While both principally work in the same way, the are referred to differently because of the way they are represented in the engine/code.

Mapping is simply the process by with 2D raster-based art is associated with 3D vector-based geometry. In this, the co-ordinates of the images must be transferred to the co-ordinates of the surface. Thus, images can be mapped to surfaces in various ways ([88]). In *projection mapping*, the texture is projected on the surface in a certain way. *Planar projection mapping* projects the image according to a rectangle, like using the image as the film in a movie projector, except that the image does not get larger when the projection surface is further away. This is generally the kind of mapping used with texturing, although each surface of a shape has its own projection. *Cylindrical projection mapping* projects the image cylindrically, like rolling the image into a cylinder

and projecting outward unto the surface. *Spheroid projection mapping* is generally the same, except with a sphere instead of a cylinder.

There are other kinds of mapping, like *lofting mapping* and *face mapping*, but the most interesting to games is *paste mapping*. Paste mapping is the mapping used in skinning. With it, each object (in this case, actor file) gets its own texture map. The texture map encompasses the entire model. It contrasts with the other mapping methods in that it does not use *tiling* of the texture (use repeated images next to each other). While it makes each object with this mapping unique, it does require more texture memory and can not handle dynamic effects very well on its own.

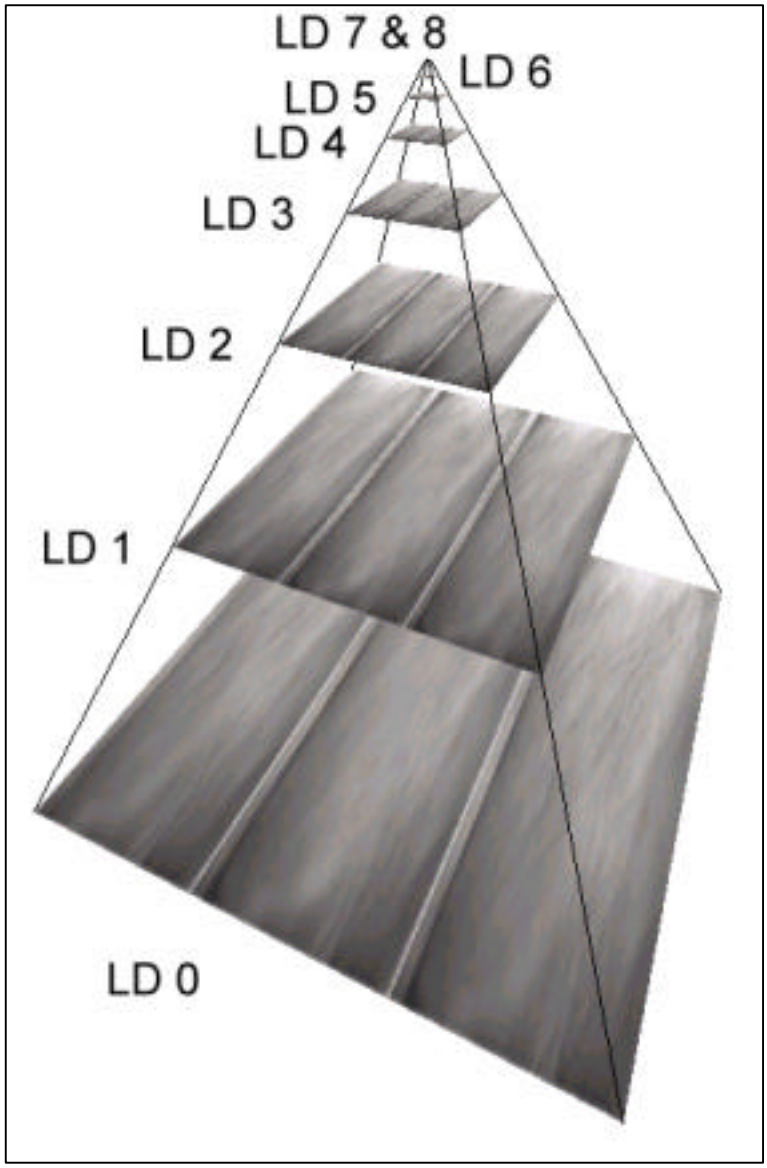
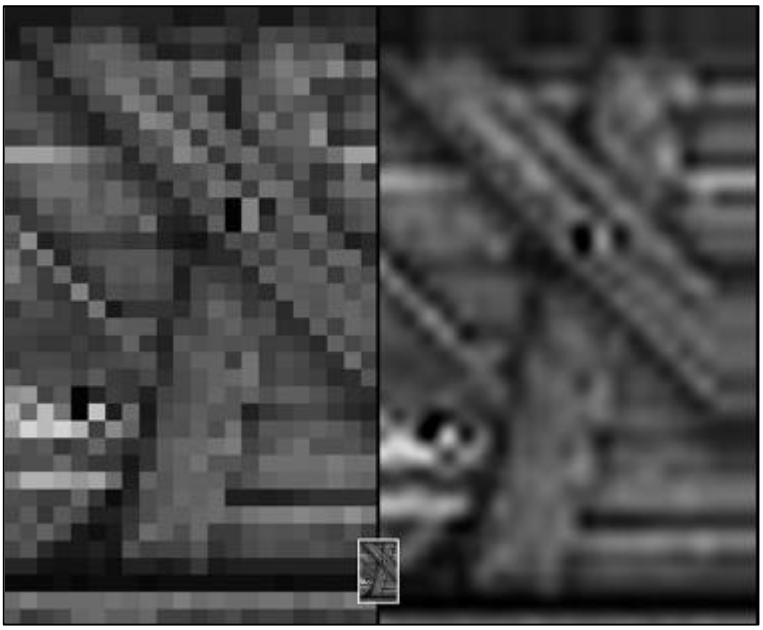


Figure 4 - Mipmapping LOD's

While this is good enough for actors, they are animated and generally detailed enough, brushes and models often require some things more. *Animated textures* are one of those things, for instance to create flowing water or a television screen. There are several types of texture animation, but they all boil down to the same thing: generating a sequence of images to replace the texture at each time interval. These images can be pre-generated, as in a cartoon, generated from transforming (moving, rotating, scaling) the original texture, or generated through some algorithm, possibly with the original texture (procedural texture animation). The first is the most common, since, although it does increase texture memory by having a short sequence of images, it decreases runtime calculation time and it can be used to generate a reasonable approximation of the other two.

Another quality one wants to assign to a material is *transparency*. Note that this is not a property of texture mapping but of how overlapping surfaces interact with each other. That being said, transparency effects in computer games are very limited. They normally do not include things like the bending of light (refraction). Still, it is better to have some transparency than none at all and in fact, transparency can be a powerful tool. A simple form of transparency is called *screen-door transparency* ([23], [53]), in which every other pixel of the surface is rendered. In this checkerboard pattern, the non-rendered pixels show the object behind it. To get a value other than 50% transparency, other patterns can be used, but such patterns are more visible. Another problem is that multiple transparent objects cannot overlap, since the front transparent object would obscure either the "transparent" pixels or the "non-transparent" pixels of the second transparent object.

A more flexible form of transparency is *alpha blending* ([53]). In this, each pixel also has an alpha (α) value, with an alpha of 1.0 meaning opaque and an alpha of 0.0 being 100% transparent. To render a transparent object (with an alpha less than 1.0) on top of an existing scene, the **over** operator is used: $\boldsymbol{c}_o = \alpha \boldsymbol{c}_s + (1-\alpha)\boldsymbol{c}_d$. $\boldsymbol{c}_s$ is the colour of the transparent object (called the *source*), $\boldsymbol{c}_d$ is the pixel colour before blending (called the *destination*), and $\boldsymbol{c}_o$ is the resulting colour. Here multiple transparent objects can overlap each other, but they are required to be sorted in back-to-front order.



Then there is *reflection*. The only kind of reflection used in computer games is *planar reflection*, as in reflection off a flat surface such as a mirror. These reflectors generally follow the *law of reflection*, which states that the angle of incidence is equal to the angle of reflection. This allows a reflection to be rendered by creating a copy of the geometry (shown in the reflection), transforming it into a reflected position over the reflective surface, and rendering it from there ([53]). Now the viewing ray does not have to be reflected off the reflective surface, but can be traced through it. Combined with transparency (of the reflective surface) this can create the illusion of a partially reflective surface, such as a very shiny floor.

Figure 5 - Nearest Neighbor & Bilinear interpolation

A problem with texture mapping appears when the distance to the texture changes. The first problem appears when the texture is magnified, or when a pixel on the texture (texel) is larger than a pixel on the screen. The two most common filtering techniques to deal with magnification are *nearest neighbour* and *bilinear interpolation*. Nearest neighbour simply sets the colour of the pixel to that of the closest texel. Bilinear interpolation computes each pixel from a linear interpolation in two dimensions of the closest four nearest neighbour texels. While bilinear interpolation is often blurrier, it prevents the blocky appearance called *pixelation* of nearest neighbour and as such is used more often.

The second problem arises when the texture is minimised, or when several texels may cover a pixel's cell. Again, the simplest way to deal with this is to just pick one texel and use its colour. This is called *point sampling* ([56]). However, this produces severe aliasing problems, and bilinear interpolation (sampling four points instead of one) is hardly better. By far the most popular method of anti-aliasing for textures is called *mip-mapping*. "Mip" stands for *multum in parvo*, Latin for "many things in a small place" ([53]). The idea is to pre-filter the original image down repeatedly into smaller images. The original texture is called level zero of the mip-map. Each successive level is one-quarter the size of the previous level, until one or both of the dimensions of the texture equals one texel. With this technique, the magnification problem can be solved by using high-resolution textures, but that would significantly increase the texture storage space.

To use this mip-map while texturing, first the *d* co-ordinate, also known as the level of detail (LOD), is computed. This is done either from the pixel's cell or from the change in the texture co-ordinate, with respect to a screen axis. This *d* is then used to determine where to sample along the mip-map's depth. The two closest mip-map levels are chosen and bilinearly interpolated samples are retrieved from each. These samples are then linearly interpolated, depending on the distance from each texture level to *d*, to get the final pixel value. This process is called *trilinear interpolation*.

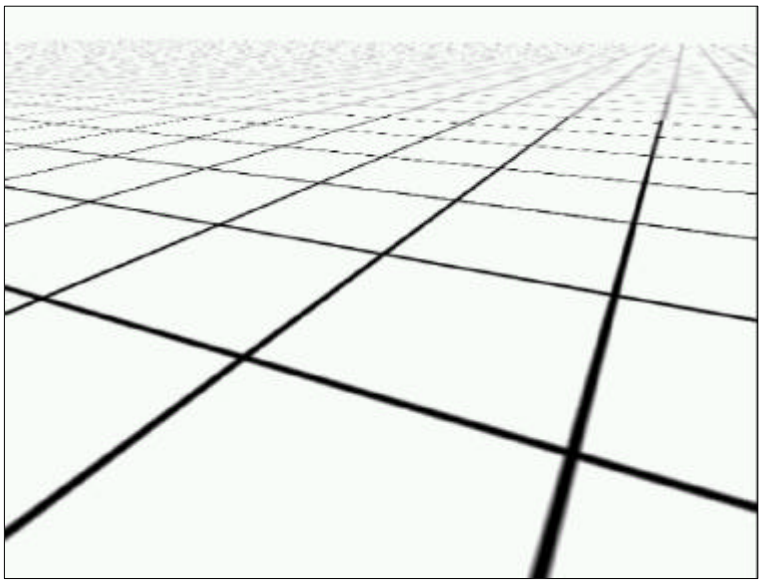


Figure 6 - Overblurring

Weaker versions of this algorithm are also used, nearest neighbour on closest level, nearest neighbour on two levels, bilinear interpolation on nearest level, but they all have the same problem, namely *overblurring* ([53]).

This is because when accessing a mip-map, only square areas on the texture are retrieved, retrieving rectangular areas

in not possible. This problem is fixed with *anisotropic filtering* ([53]) algorithms (rip-maps, summed area tables), which can retrieve texel values over areas that are not square. However, they do this most efficiently only in primarily horizontal and vertical directions and are memory intensive. A relatively new anisotropic filtering method that uses a line of anisotropy does not have these problems, since it can run in any direction and uses the mip-map algorithm to do its sampling.

2.3 Light and Shadow

With objects and textures applied to them, things should be looking better. Nevertheless, there is still no way to see anything (or everything is full bright). What is missing is *light* and *shadows*. Lighting makes sure there is something to see and gives objects depth. Shadow indicates where there is no (or just less) light and gives a scene depth. Light and shadows are the final ingredients needed for creating a realistic 3D picture.

There are a few ways to light a world ([23]). First and easiest is *ambient light*. Ambient light produces a constant amount of light on all surfaces. In more advanced lighting models it is used to mimic light reaching a surface by bouncing on other surfaces, or *indirect lighting* (see 4.3). Ambient light is globally set and thus no entities are used to set it. The amount of ambient light is set high if one wants to view the geometry, but in lighted environments the amount of ambient light is usually set low or even turned of. Ambient light can also be coloured.

The second type of light is *positional light* ([53]). Positional lights have a single point from which their rays emanate. Two kinds of positional lights are *point lights* and *spotlights*; point lights emanate light rays in all directions and spotlights emanate light in a certain direction with a cut-off angle. Positional lights are placed in the world with an entity, whose location represents the source point. The rays usually have a strength which diminishes over distance, which is determined by a light value (higher value means greater distance). Positional lights are often the only light types used to light a world in computer games (although ambient light is often included).

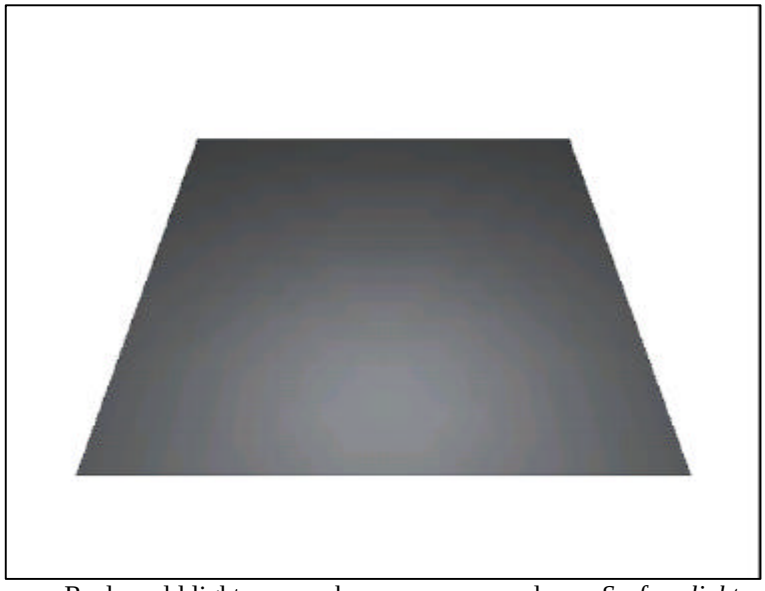


Figure 7 - Shading from point light

The third light type is *directional light* ([53]). With directional lights, the rays do not come from a single point, but from a single direction. This is the same as an infinitely distant positional light. How much effect the light has on the surface depends on the surface's orientation. If the surface is perpendicular to the rays, it is brightly illuminated; the more oblique the surface is to the light rays, the less it is illuminated. Note that if the surface faces away from the (infinite) light source, it should not be illuminated at all. Amsterdoom does not have directional lights (its Sun entities are a weird case of point lights).

Real-world light sources have an area or volume. *Surface light* comes closest to representing this. In it, the surfaces themselves emit light. The colour used is usually the average colour of the surface. Even in games that support them, surface lights are used sparsely and positional lights are used instead. In those cases, surface lights are used to create the illusion of glowing surfaces (such as lava). Amsterdoom does not have surface lights, but instead uses positional lights to simulate the effect.

All these lights affect the objects in the world. The interaction between material and light sources, as well as their interaction with the geometry of the object, is called *lighting*. The casting of *shadows* might be seen as part of this. *Shading* however is the process of performing lighting calculations and determining pixels' colours from them. There are three main types of shading ([23], [53]): *flat shading*, *Gouraud shading*, and *Phong shading*. It is important to understand that surfaces are usually divided into triangles.

With flat shading, a light value is calculated for each triangle and that light value is applied to each point of the triangle. This is simple to implement and fast and thus used very often. However, it does not give a smooth look to approximations of curved surfaces as the other shading methods do. Gouraud shading calculates the light intensity at the corners of the triangles and interpolates those values across the surfaces of the triangles. Gouraud shading is still fast and its quality is an improvement over flat shading. A problem with this is that the technique is highly dependent on the level of detail of the objects that are illuminated. In games, Gouraud shading is used to give objects a round look. Phong shading improves this by interpolating the surface normal and computing the lighting at each point. This is a very complex and costly method and as such rarely (if ever) used in games. This might change in the future.

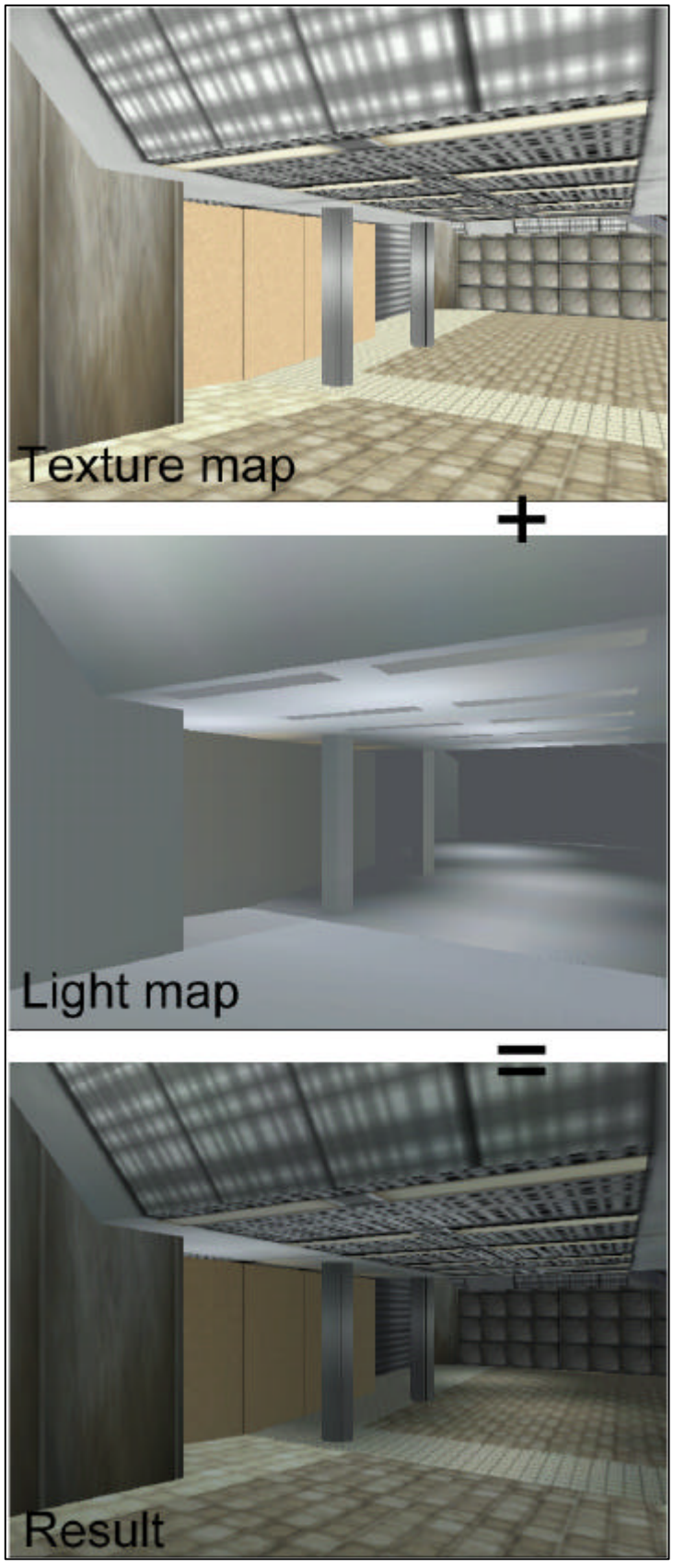


Figure 8 - Light mapping

The light values above are computed using a *lighting model*. In real-time graphics, these models are very similar and can be divided into three parts, namely the *diffuse*, the *specular*, and the *ambient* components. The diffuse component determines how much light a surface reflects back based on its orientation and the direction of the light. However, this is based on ideally diffuse (totally matte) surfaces. To make a surface look shiny, the specular component is used. This component makes highlights appear on the surface. Finally, the ambient component is the effect of ambient light on the surface, or more correctly the light reaching the surface not directly from the light source.

Where shading gives objects depth, shadows give a scene depth. Shadows result when light falling on an object is partially blocked by another object. The object on which the light falls is called a *receiver* and the object that casts the shadow is called an *occluder* ([53]). The area where light is blocked is the shadow. Shadows are generally pre-calculated. Even dynamic lights that are able to cast shadows off static objects pre-calculate the possible values. Actors and other dynamic objects are either not given a shadow, or a shadow is implied by placing a blurry dark spot on the ground under the object. This is by no means a real shadow, just something to give the impression of one. However, real-time calculated, dynamic model shadows are starting to be used. For more on shadows, see section 4.2 on dynamic shadows and section 5.2 on soft shadows.

While Amsterdoom can do flat shading and Gouraud shading, its standard shading method is something else. Actually, it, and many games with it, uses a *light map*. A light map is a separate, precomputed texture (see 2.2) that captures the lighting contributions ([53]). By multiplying it with the underlying surface (the texture map), one can achieve Phong-like shading. Light mapping should actually be termed “dark mapping”, since the original surfaces are actually decreased in intensity.

Light mapping requires an extra rendering pass, but it pays back this additional cost with a number of advantages ([53]). Light textures can generally be low-resolution, since light often changes slowly across a surface. Light textures can be swapped on the fly with minimal processing. Lighting conditions can be reused and a normal texture can have different lighting conditions at different locations (as opposed to pre-multiplying light texture and normal texture beforehand). Light textures can be animated easily, or even be recalculated on the fly. On top of that, with *multitexturing* support in hardware, even the added cost disappears.

With light mapping and texture mapping, one has two passes. However, more passes are possible, resulting in *multipass rendering*. In this, each successive pass modifies the result from the previous one. Four texture operations are commonly used in multipass rendering: replace (replace previous with new), modulate (multiply textures), add/subtract (add/subtract textures), and blend (use alpha value). Often the same geometry is reused in each pass and can be used for a wide range of results such as bump mapping, diffuse lighting, texturing, specular lighting, and atmospheric effects. Quake 3 uses ten passes ([53]). Dynamic lights are often pre-calculated and are realised by swapping the light map.

2.4 Others

With the ability to create realistic looking images, there is still not much happening. To get something going in a computer game, *animation* is needed. Animation is more then just a sequence of images; it introduces dynamics to a computer game. It involves everything from moving the camera through a world, to animating characters and objects, or showing a cut-scene animation sequence (a pre-rendered sequence of images in between two sections of gameplay). Then there are the extra ‘special’ things that can be included in a game to make it more spectacular, the *special effects*.

With the real-time nature of computer games, animation is an important aspect. In fact, without animation pre-rendering a scene would be enough. An important term in animation is *keyframing*. A keyframe is simply a position for the animation to reach, comparable with a frame (a single image) in a 2D animation sequence. Keyframes can be completely hand-created or they can be generated from a simulation algorithm, or possibly a combination of the two. Depending on the type of animation there might be interpolation between keyframes. Different types of animation can also be combined (and usually are).

Of the several basic types of animation ([88]), *camera motion* is the simplest. Here the viewpoint (often referred to as the camera) is the only thing moving. The data for this animation is usually provided from the user’s input, especially in games, where a large part of the interactivity comes from this kind of animation. Of course, it is very possible to pre-set camera positions and directions and move the viewpoint (through interpolation) to those positions. This is generally used for so-called in-game animations, which are animation sequences using the game engine over which the player has no control (like a short movie within the game). They are usually coupled with another type of animation (as an example, look at Unreal’s start-up animation). Another use for this is placing cameras in the game so that the users can look at one part of the environment while physically in another part (security cameras).

Other types of animation are *object movement* (which is a more general case of model animation discussed in 2.1) and *texture animation* (which has already been handled in 2.2). *Hierarchy animation* is similar to object motion in that the objects can only move relatively to each other, but not move relative to themselves. The difference is that the objects are connected in a hierarchy, which constrains their motion in logical ways. The pieces usually move close to each other. Sometimes they are even intended to touch or intersect. A somewhat different type of animation is *property animation*, in which the properties of objects change. This is most apparent in objects like dynamic lights, where the change of radius and colour form the animation.

In *vertex animation*, also known as 3D morphing or vertex-level manipulation, points (the vertices) of an object move relative to the rest of the object. This changes the shape and thus, if done correctly, the posture of the object, so that for instance a “walking” animation can be done. In computer games, this type of animation is generally reserved for actors. The various postures of the actor are placed together in a sequence of pre-generated keyframes. During animation, a selection of this sequence (the “walking” part or the “jumping” part for instance) is shown with generally no interpolation between the frames. Note that the skin is normally kept separate from the keyframes, so that each frame can be skinned easily. Real-time vertex manipulation, in which the positions of the vertices are calculated during runtime, is just starting to be used. It is discussed in section

5.1, together with the last type of animation, *skeletal deformation* (*bone animation*), which is another way to exert influence over the vertices and thus move them.

In computer games, intelligent actors (particularly the enemies) often use a *waypoint system*. In some ways, waypoints are very similar to keyframes, in that they serve as location targets. However, waypoints are generally not traversed in a set sequence and thus leave the object a certain amount of freedom. While a strict waypoint system (such as Amsterdoom uses) does not allow the actor to deviate from the path defined by the waypoints, other systems do allow this. Waypoints may also have additional information, like light value and command changes, which tells the creature to use another decision-making algorithm that, in turn, might generate another animation sequence.

Physics also plays an important part in animation and manifests itself in two ways. First, it is apparent in movement, where gravity might affect jumping and falling, and friction might affect walking and gliding. Second, it is apparent in animation sequences, where a heavy creature walks different from a light creature. Physics does not only affect the player and other creatures, but every object that can animate.

Special effects are not necessary to create a great game, or improve realism. They are generally thought of as icing on the cake, and, in specialised circumstances, they do improve realism. Currently, there are several special effects commonly used in computer games.

Particles are small objects that are set into motion using some algorithm ([53]). They are a method of animation and usually have a relatively short lifetime. One form of a particle is a single point rendered on the screen and another is a line segment from the particle's previous location to its current one. Particles can also be linked with (usually small) actors to simulate things like flying debris. Particles are used to simulate things like fire, smoke, explosions, impact debris, and similar phenomena.

Decals are 2D, static pictures placed on other surfaces. They are a method of making changes to textures without actually changing the texture. Benefit of this is that they are independent of the textures and their location, but each decal also adds another surface to be rendered and with many decals, this can impede on the rendering speed. Still, they increase realism because they add persistence to the world. Decals are used to simulate things like scorch-marks and blood spatters.

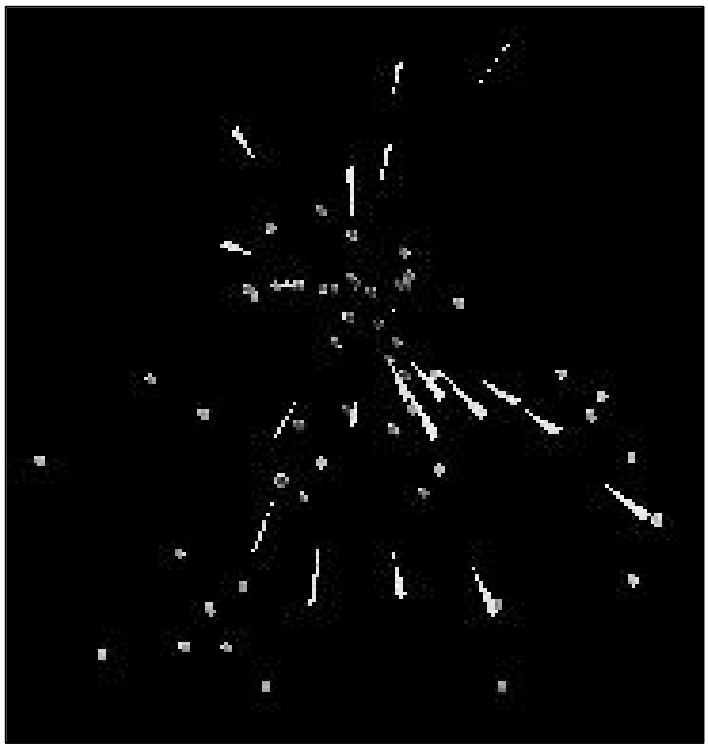


Figure 9 - Particles

Billboarding is simply the technique of having a two-dimensional surface, placed in the 3D world, always oriented based on the view direction of the observer. Billboards, the polygons that show this behaviour, can be used to represent many phenomena that do not have solid surfaces. This includes smoke, fire, explosions, etc. Billboards can be full-screen. This can induce the effect of viewing a scene through night goggles by superimposing the billboard over the screen. By placing the billboard behind everything in a scene, environment outside the geometry can be simulated, for example the sky. By changing the co-ordinates at the corners, the sky can be made to appear to rotate with the view change.

Lens flares are caused by lenses, of either the eye or a camera, that are directed at bright light ([53]). They consist of a halo, caused by the refraction of light of different wavelengths, and a corona, caused by density fluctuations in the lens. The halo appears as a ring around the light and the corona appears as rays radiating from a point, which may extend beyond the halo. Camera lenses may have secondary effects. The use of lens flares in games is often complained about with phrases such as: "I don't walk around looking through a camera." However, people often forget that the eye has a lens too and that, when looking at bright light, a flare does appear. More often, what they actually complain about is the overuse of lens flares and them often being too strong. In computer games, lens flares are created by placing an entity before the bright light. During the game, this entity represents the spot to place one or several screen-aligned images (billboards).

Figure 10 - Lens flare

Environment mapping is a technique created for reflections and this *reflection mapping* is further discussed in section 4.1 of this paper. The only use of an environment mapping type technique currently used in computer games is *cubic environment mapping*, and is used to create the illusion of space outside the geometry (such as the sky). The technique consists of projecting an environment on the sides of a cube. This cube is then shown instead of specialised surfaces (with textures marked sky) with the viewpoint as the centre of the cube and irrespective of the orientation of the surface. Many games create this environment map on the fly with a so-called skybox. This skybox is a section kept separate from the rest of the geometry and represents the environment to be mapped. The benefit of this is that it can change during run time, which makes the environment map dynamic. The cubic environment map is used together with billboarding techniques.

Fog is an effect that is not only used to add realism to outdoor scenes and help viewers determine the distance of objects, but can also help mask limitations of the engine. Objects beyond a certain distance, called the *far plane*, are not rendered (in fact, they are clipped by the far plane). Fog can help mask this effect, by letting objects near the far plane be invisible due to thick fog. This way, objects seem to fade away in the distance. Fog is usually added to the final image and thus can be rendered in a separate pass, if care is taken.

Chapter 3 – Future in Objects

Computers have a limited amount of triangles (or rectangles) they can show in one screen while retaining an acceptable rendering speed. Part of the need for faster hardware (both processor and graphics) is to increase the amount able to show. Yet, the limitations exist, also limiting the amount of detail one is able to show. To circumvent these restrictions, techniques have been developed to scale the amount of detail depending on requirements, or to create the illusion of detail where there is none.

3.1 Curved Surfaces

In every computer game until now, surfaces consisted of either triangles or rectangles. While they are quite versatile and can be used to build many shapes, they lacked curvature. They could be used to approach the curved appearance of, for instance, a sphere, but many sections were needed to loose the sharpness created by the connection of flat surfaces. Approximating round surfaces this way also requires a lot of storage space. Using many sections would also increase rendering time, making only the best machines able to handle them and thus creating a lack of scalability. To create truly round shapes, flat surfaces simply do not suffice. *Curved surfaces* are what one needs for that.

The first problem is how to define these curved surfaces or, more basic, curved lines. There are several types of splines, which for all practical purposes, simply is a curve. First, there is the *natural spline*. A natural spline is defined by control points that lie on the curve itself. Moving these control points changes the curve. Natural splines are very limited and more control points are needed to generate more subtle curvature.

More control points is exactly what Bézier curves offer. In addition to the two control points that actually sit on (the section of) the curve (interpolate the curve), there are two control points (per section) that do not lie on the curve. These points, often visibly connected to the interpolating points by lines, can be moved about to define the curvature. Bézier curves are commonly used in 2D drawing programs.

To have even more control over the curvature, *B-splines* can be used. B-splines do not require the control points to lie on the curve. This makes the shaping of B-splines feel somewhat like a puppeteer manipulating a marionette. Moving a control point only affects a small part of the curve, which is called *local control*. A B-spline consists of m control points ($m \geq 4$) and m-3 curve segments. The endpoints of each section are called *knots* and there are always m-2 of them.

A special kind of B-spline is the non-uniform, rational B-spline, sometimes called a *NURBS*. With NURBS, one can change the “weight” of control points, thus changing their impact on the curve. This allows one to edit the curve without moving the control points. NURBS can also be cut more easily, namely anywhere along the curve, where B-splines can only be cut at the knots.

A flat surface, also called a *polygon*, is bounded by straight lines. By bounding a surface by curves, one has a curved surface, also called a *patch*. While any type of curve can be used to define a patch, here the focus will lie on Bézier patches, since that is the kind talked about most often in relation to real-time graphics. NURBS

surfaces and B-splines in general are more flexible, but at the cost of increased computation and modelling complexity ([44]).

A Bézier patch is defined by sixteen control points, four for the corners, eight more for the bounding curves, and the last four for the surface. Note that surfaces with three corners is possible by shrinking one of the four boundary curves to length zero and overlapping the two corner points. Besides allowing easy control, Bézier patches are also contained within the convex hull of their control points, which is useful for coarse collision detection. The convex hull of the control points is what you get if you stretch a rubber sheet over the control points. To connect patches, they must share two corners, but also the two control points along the curve connecting the two corners. Still, a crease might be visible between the two patches.

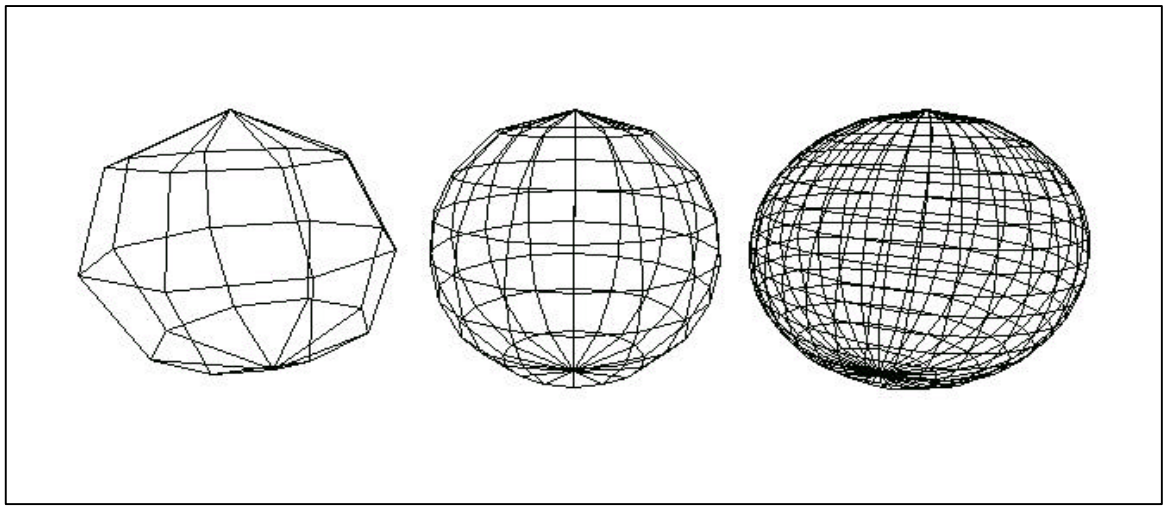


Figure 11 - Tessellation

To be efficiently displayed on current graphics systems, curved surfaces must first be approximated with triangles ([45]). The process of converting a curved surface to a triangle approximation is called *tessellation*. While this can be done statically, all the benefits of using curved surfaces would be lost; there are still large storage requirements and there is still a lack of scalability. Tessellation can best be done dynamically. Dynamic tessellation allows for the inclusion of a variable stating the bound on the triangle size. This way, low-end systems can allow bigger triangles, thus decreasing the number of surfaces, and thus decreasing rendering time. Tessellated curved surfaces also work well with static levels of detail (see section 3.2) to result in significant model size reduction and rendering speedup.

Ray tracing and rendering software usually display curved surfaces by tessellating them first. This does not have to happen dynamically, since pre-rendering systems do not have to be very scalable (it just takes longer to render an image on low-end systems). The triangle size bound is still a possibility, to balance between rendering time and quality of the curved surface. Further, NURBS surfaces are the most common type of curved surfaces in this situation; even tough they often behave in much the same way as Bézier surfaces.

Figuur 4 - Quake 3 - Arena

Computer games do not have curved surfaces yet. The static representation of geometry has been good enough and implementing curved surfaces was just too expensive. Yet, a few upcoming games including id Software's *Quake III: Arena* and Shiny Entertainment's *Messiah* are incorporating curved surfaces. Id Software has always been the leading voice in 3D, first person, computer graphics, so the consensus is that curved surfaces will become standard in the future of computer games.

One must wonder how accurate this assessment is, however. Increasingly voices are crying for an increase in speed over an increase in visual appearance, and curved surfaces add little in terms of

gameplay. So what do curved surfaces add? They can add an incredible amount of realism. Just look around, there are few environments without at least some curved surfaces, making it seem paramount to include them. Adding them to computer games makes them appear that much more realistic and beautiful.

Computer games do not need curved surfaces to look great visually, however. Just look at Amsterdoom, where it does not try to simulate reality, it nearly prides itself on its rugged, unbent appearance. Even where it does try to simulate an existing scene, it does a fair job of approximating curvature. Lack of curvature will not be noticeable during gameplay anyway. Yet, it is undeniable that curved surfaces would add to the realism of scenes that include approximations of curvature. While it is not noticeable now, it will be in the future, as more games start including curved surfaces by default. Players will get accustomed to seeing curved surfaces and will expect them, which will make any game that does not have them seem lacking.

In conclusion, while visually realistic computer games can and will be made without curved surfaces, curvature will become standard in more and more future computer games, until they are included in games by default.

3.2 Levels of Detail

In three-dimensional scenes, every geometric object is typically modelled using a single, fixed resolution model. This can be crude for nearby objects, where the chosen detail level is insufficient, or it can be slow for distant objects, where the detail level is excessive. What is needed are multiple *levels of detail*, so that low detail can be chosen for distant objects while high detail is chosen for near objects.

A levels of detail technique has already been given above, for textures, namely mip-mapping (see 2.2). In it, a texture consisted of various levels of detail for various rendering sizes. The problems textures had also occur with models (here the term models is abused for brushed, models, and – mostly – actors). Say that a model does not take up more than one screen pixel due to being far away. With a *fixed resolution model* (where the level of detail is fixed) ([31]), the model still has to be rendered completely and calculation time is wasted on a single pixel. However, if low detail models are used to speed up rendering of distant models, the models look crude when viewed up close. A resolution somewhere in between has the worst of both worlds, though to a lesser extent.

Multi-resolution modelling offers a technique to handle these two problems. The idea is to use simpler versions of an object as it gets farther from the viewer ([53]). The increased distance makes sure that the simplified version looks approximately the same as the more detailed version. Some objects, like curved surfaces, have a level of detail as part of their rendering description, which determines how they are dynamically tessellated into displayable polygons.

The first possibility to achieve levels of detail is to have different representations simply be models of the same object containing different numbers of primitives, or *discrete geometry levels of detail* ([53]). This is similar to the simplest mip-mapping technique. Depending on the distance from the viewer the model for the correct level of detail is chosen. These models may be hand-created or pre-generated using various simplification algorithms. There are a few drawbacks to this approach to levels of detail, though. One problem is that *popping*, a noticeable, distracting, and abrupt model substitution, effect may occur when switching between detail levels ([53] and [34]). Blending can be used to avoid this problem, but during the transition, two levels of detail have to be rendered and blended, which costs time. Another problem with this solution is that it requires a lot of memory.

The simplification methods used to pre-generate the models for the various levels of detail can also be used to create the various levels of detail from a single complex object, or *geomorph levels of detail* ([53]). The idea is to form dynamically the appropriately detailed model according to the distance from the viewer. To prevent the popping effect, the transition from one model to the next is achieved not by blending, but by moving the vertices to the location on the next model by interpolation. A disadvantage is that simplification can give rise to disturbing artefacts.

A technique very similar to this is Intel's MRM (Multi-Resolution Mesh) technology ([34]). MRM consist of a single, high-vertex-count model, plus a set of instructions that allow vertices to be removed or replaced, one at a time. Since the changes are so small, they are not visible, yet the complexity of the model is reduced without the popping effect. Although an MRM model is typically twice as large as a fixed-resolution model of the same resolution, it is notably smaller than what you would get with discrete geometry levels of detail. Of course, any such automatic process must take important features, like fingers on hands and the shape of a head, into account.

Another technique is the use of *alpha levels of detail* ([53]). As the distance of the object increases, its overall transparency increases with it (a is decreased) and the object disappears on reaching full transparency. The benefit of this technique is that the object continuously changes and is finally not rendered at all, but the disadvantage is that the object disappears.

Because levels of detail techniques are used to speed up rendering, and they allow models that are more detailed, they are of little use to pre-rendering ray tracing. The techniques might be of some benefit in reducing compute time, especially for 3D animation, but their main strength lies with application in the real-time field.

Figuur 5 - Vampire, the Masquerade - Redemption

Computer games are starting to catch on to the advantages of levels of detail. They have embraced mip-mapping and are likely to do the same with multi-resolution modelling. Intel's MRM technology is already being incorporated in future games as Valve Software's *Team Fortress 2* and Activision's *Dark Reign 2*, and similar advances are used in Shiny Entertainment's *Messiah*, Monolith's LithTech engines ([55]) and Nihilistic software's *Vampire, the Masquerade - Redemption*. Since the technique seems so beneficial for so little cost, it is expected that more games will incorporate the technology. Unfortunately Amsterdoom does not do so.

Of the other levels of detail techniques, none are very likely to be used much. The popping effect occurs with discrete geometry levels of detail, a problem that the other techniques do not have. With alpha levels of detail, the problem arises that objects do not really become less detailed, the just become less visible, and the removal of objects is something games do not want (in this manner). Geomorph levels of detail might be the only other technique used, a technique like this might be used in Messiah and the LithTech engines, but the interpolation of vertices might cost too much in comparison with the other alternative.

Levels of detail techniques are not ray tracing techniques, but are used to bridge the gap between real time rendering and ray tracing. Thanks to levels of detail, computer games can achieve ray tracing quality models before reaching the computing power needed to make ray tracing real-time. The techniques might slowly disappear when real-time ray tracing becomes possible, but since that is probably many years away they will become standard techniques for quite some time.

3.3 Displacement Mapping

Textures can convey the feeling that a surface has material properties. Surfaces can look like wood, stone, dirt, cloth, or any other type of material. In fact, Amsterdoom uses these qualities to great effect. Nevertheless, surfaces always appear to be flat and it takes a very good texture to make surfaces seem somewhat bumpy. What is needed is a technique that gives surfaces a bumpy appearance. Maybe even have several features stick out (like a painting on a wall, which is normally just a piece of the texture and which one wants to stick out). Several techniques exist that accomplish this, namely *bump mapping* and the more advanced *displacement mapping*.

Bump mapping is a technique that makes a surface appear uneven in some manner. This is realised by changing the surface normal used in the lighting equation. The surface itself remains smooth in the geometric sense. Just as having the same normal for pixels gives the illusion that the surface is smooth (shading, see 2.3), modifying the normal per pixel changes the perception of the polygon surface itself.

There are two basic methods of modifying the normal with a bump map. The first method uses two values that correspond to the amount by which to vary the normal along the image axis. The second method uses a *height field*, which is a monochrome texture value that represents the height: white is high and black is low, or vice versa. This height is used to calculate the slope between points, which is used to calculate the direction of the normal. Both methods are convincing and an inexpensive way to add geometric detail.

The weakness of these methods lies in the fact that the surfaces only appear bumpy. When looking at bumped surfaces from the side (silhouettes of objects) the viewer sees that there are no real bumps, just smooth outlines. Mip-mapping also does not work with bump textures, since the effects of bumpiness disappear. Shadows are also not produced by the bumps, which can look unrealistic. Another drawback is that almost no real-time systems can handle a per-pixel varying normal in lighting equations. However, with respect to this last problem, techniques exist that can be used for real-time bump mapping.

Instead of storing height or slopes, one can actually store the new normals as vectors in a *normal map*. Although Direct3D supports this specialised texture-blending operation, there is no widespread hardware support for it. Another method is more popular, as it uses commonly available capabilities, though at the cost of additional rendering passes.

Figure 12 - Embossing

This other method uses a technique borrowed from two-dimensional image processing and is called *embossing*. The chiselled look it gives to an image is achieved by copying the height field image, shifting it slightly, and subtracting it from the original. This same technique can be used with 3D surfaces by shifting in the direction of the light. This is done after the surface is rendered with the height field applied. The third step is to render the surface again with the height field, subtracting it from the first-pass result. Finally, the surface is rendered again with no height field, simply illuminated and Gouraud-shaded. Step three gives the emboss effect.

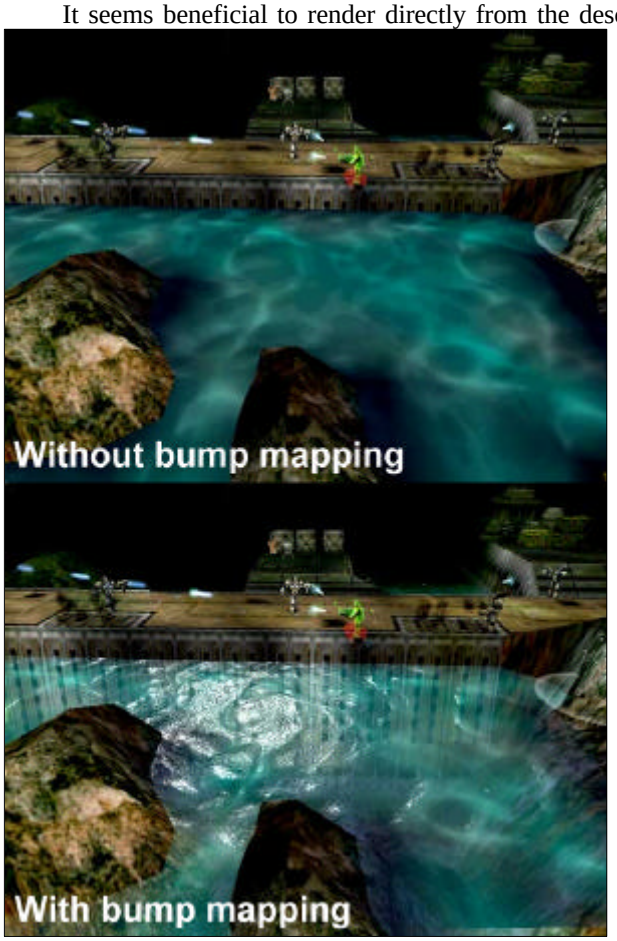
Other methods can be used for Gouraud-shaded bump mapping. One method is to distort environment mapping (see section 4.1) by differentials found in the bump map. This gives the effect of wobbling the look of the reflective surface. This technique is tricky to control and expensive to implement in hardware (the Matrox G400 is the first chip known of to support this technique). Benefits are that it requires only one additional pass and supports any number of lights.

Where bump mapping only creates the illusion of elevation changes in a flat surface, displacement mapping actually performs the geometrical displacement of the base surface ([32], [26]). This allows for effects such as self-occlusion (part of the surface obscures part of the rest of the surface) and self-shadowing (surface casts shadows on itself). It also results in more realistically looking images, in particular in silhouette regions.

There are several ways to realise displacement mapping ([26]). One is to vary the pixel position on the screen, but it cannot handle large displacements and view dependent sampling, which is important to represent the silhouette. A second approach is to resample the displacement in volumetric textures. However, due to the increased storage space this approach is restricted to either repetitive textures or small displacements. On top of that, the illumination requires either increased computational costs or increased storage space.

Another possibility is to tessellate geometric objects with displacement maps into a multitude of small triangles ([32]). In this approach however, the resolution independence is lost and it is not very storage efficient. Furthermore, it is possible to miss narrow features, which is particularly disturbing if the displacements of those features are large. Finally, numerical errors can occur in intersection tests with many small polygons.

It seems beneficial to render directly from the description of displacement maps, without generating any intermediate geometry. In [32] a method is proposed for ray tracing based on the idea of hierarchically subdividing the parameter domain of the surface. It tests intersection with the bounding box of the displaced surface. If intersection occurs, it subdivides the surface into (four) sections to recursively test intersection with each of the sections' bounding boxes. Intersection with the surface is detected by having a small enough bounding box. Computation time can be traded for memory to increase performance.



Problem with this method is that it might not be suited for real-time application. Real-time systems might benefit more from a dynamic tessellation technique, similar to that for curved surfaces (see section 3.1). More research needs to be done in this area.

Figuur 6 - Matrox G400

Displacement mapping is currently not used in computer games. As stated above, more research needs to be done to make displacement mapping possible in real-time environments. Problem is though, that it either increases the polygon count, something that slows the rendering process, or slows the rendering process directly (by performing a multitude of intersection tests for just a single surface).

Bump mapping however is being used in some games. For instance, Appeal Software's *Outcast* (which is voxel based instead of polygon based), and Matrox G400 enhanced games like Rage Software's *Expendable* and Outrage Entertainment's *Descent 3*, support bump mapping. Still, unless more graphics cards start supporting it, bump mapping is not going to be a widely seen feature, let alone a standard technique.

As for the future of displacement mapping, this will stay out for quite some time. First bump mapping has to become more standard, which does not seem to be a priority of many developers. Then there is the distinct possibility that displacement mapping will be passed by in favour of using higher polygon counts to increase detail in the environments. Features that stick out will be modelled separately, they might even become dynamic, and bump mapping will be used to create the material specific depth (irregularity of stones, wood, dirt, etc). Displacement mapping might be used to ensure that in silhouette, the surfaces still appear with the correct depth, but only minor height variations will be needed for that.

3.4 Objects Conclusion

Objects do not make the world go round; they *are* the world. Without objects, there would be nothing to display, nothing to interact with, nothing at all. This chapter looked at more advanced ways to represent 3d objects with as a goal to build more realistic worlds. *Curved surfaces*, *levels of detail*, and *displacement mapping* have been looked at.

Using objects that are more advanced usually comes down to finding a way to use more detailed objects. Theoretically, everything could be built with a lot of incredibly small, triangular polygons. Such a representation would be highly impractical, however. This is why most of the discussed techniques focus on dynamically adding detail to an environment.

This dynamic addition of detail has a few disadvantages, however. First, the detail actually has to be generated, adding to calculation time. Because it has to be dynamically generated, pre-calculation techniques can be very difficult. How does one know how to light something if one does not know exactly how it will look yet? Another problem is the switch between various amounts of detail. This can be very noticeable, so the amount of detail has to be kept static, losing most of the advantages of dynamically added detail, or great care has to be taken, which might result in more calculation time needed.

Luckily, the advantages generally outweigh the disadvantages. The representation can be very compact, where only a few parameters can generate any number of intermediate values. This is the case with procedural geometry, mostly. The second advantage is that it is very scaleable. Not only can far objects be less detailed than near objects, but also the amount of detail used can be tuned to the system specification, so that al systems run at maximum amount of detail. Lastly, it allows for very detailed objects, while maintaining rendering speed.

There are other ways to increase detail. Some methods generate the illusion of more detail. Textures are an example of this, but they are still very limited. A few of these limitations can be lifted with bump mapping, but this still has its problems. Higher resolution textures would be another way to increase detail, but this also increases memory requirements a great deal.

Various techniques to achieve various differing results have been discussed in this chapter. Section 3.1 described techniques to include curved surfaces in an otherwise flat environment. Section 3.2 described levels of detail techniques that enable objects to be more detailed, while maintaining speed. Section 3.3 described bump mapping and the more advanced displacement mapping, allowing otherwise flat surfaces to have dimension.

The tendency to include curved surfaces in computer games comes from the observation that in the real world many surfaces have curvature. Where the tendency has been to create round objects (like pillars and spheres) with many surfaces and a (Gouraud) shading algorithm, this is neither efficient nor practical. Bézier patches are the most likely candidates for curved surfaces in computer games, simple yet versatile, but they may be upgraded to NURBS surfaces sometime in the future. The curved surfaces are dynamically tessellated into triangular approximations of the curvature, so the surface can be efficiently rendered. This tessellation works very well with the in section 3.2 described levels of detail techniques. Although computer games do not necessarily need curved surfaces, it is very likely that they will become standard in the industry.

Levels of detail techniques are techniques used mainly to speed up rendering of detailed objects. This is done by removing detail (polygons) as less detail is needed. This can be achieved by taking representations of varying detail, but this does not only require lots of storage space, but also has a very noticeable popping effect when changing from one representation to another. Intel's MRM technology and similar technologies are the most promising levels of detail techniques. If real-time ray tracing becomes a possibility, levels of detail techniques might no longer be needed, but that is a long way in the future.

Where bump mapping only appears to give a flat texture 3D properties, displacement mapping actually does make a texture 3D. The benefit of bump mapping is that it is relatively simple without truly adding detail, but the lack of depth becomes apparent when seen from the side. A way to do displacement mapping is by

dynamically tessellating the geometric displacements, but might require too many polygons to be convincing. Thus, it might be better to use levels of detail techniques rather than displacement mapping. Displacement mapping will not be used soon in computer games, at least not until bump mapping becomes standard.

The techniques described in this chapter enable a higher level of detail than would normally be possible. The techniques can add scalability and look far more realistic. The techniques are not pure ray tracing techniques, but instead ray tracing uses them to speed up rendering of complex scenes. They imply that near ray tracing quality images might be achieved in real-time long before actual real-time ray tracing is possible.

Chapter 4 – Future in Light

As far as light is concerned, the two virtues of beauty and realism are inseparable. This is because, to the human eye, light is inherently beautiful. Creating physically accurate lighting in the graphics of computer games thus gives a certain satisfaction beyond creating something accurate. This chapter looks into various aspects of light, including reflection and transparency, shadows and indirect lighting. This is a continuation of lighting as discussed in section 2.3, aiming more towards future techniques.

4.1 Transparency, Reflection and Refraction.

When looking though glass in Amsterdoom, one looks straight through to the geometry behind it. This is not very realistic. Not only is normally the image through the glass slightly shifted (*refraction*), but the glass also shows a mirror image of the geometry in front of it (*reflection*). Moreover, the effects are not just restricted to glass. Water, plastic, etc have similar properties where many others, like mirrors, polished wood, metallic surfaces, etc, have just one. By including these properties one could increase realism.

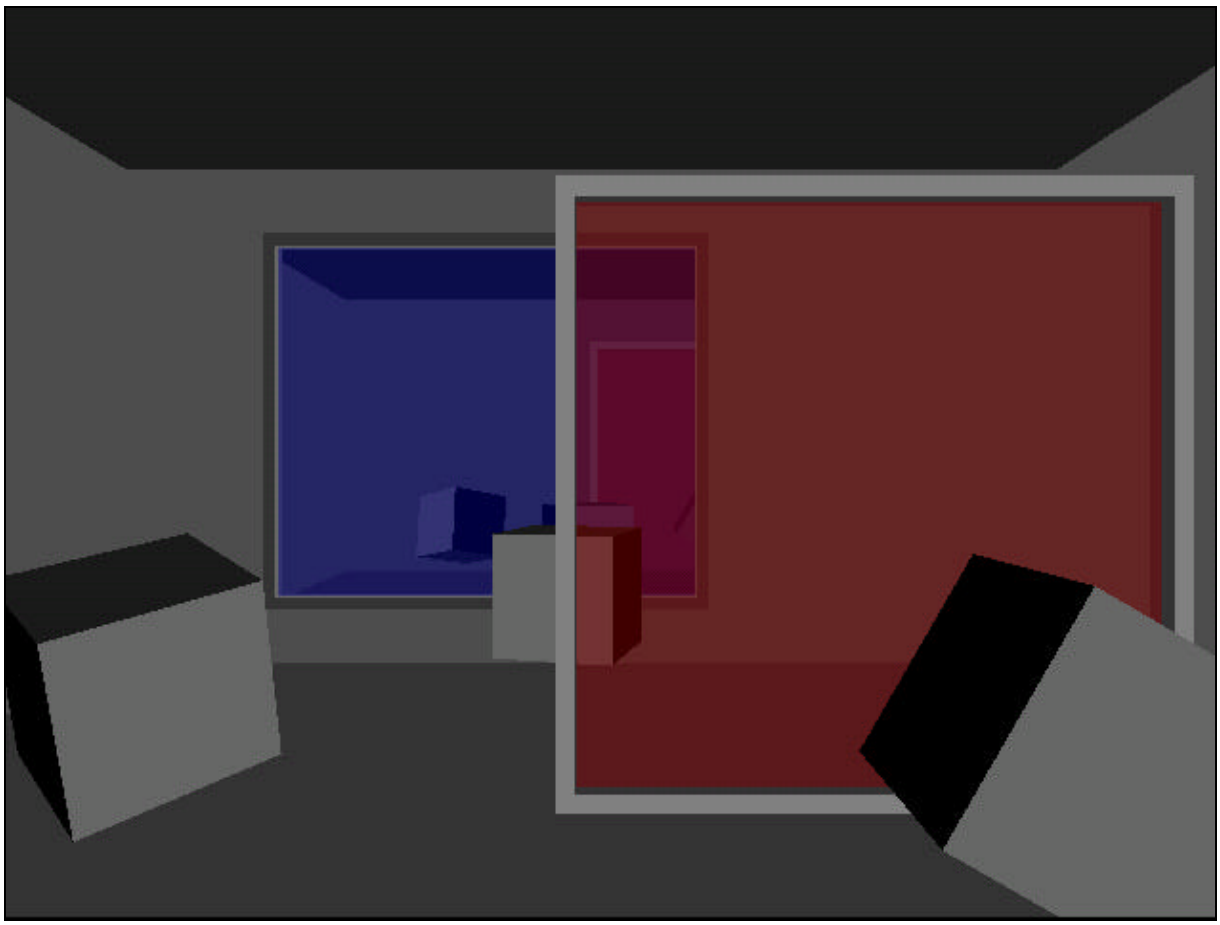


Figure 13 - Transparency and reflection in Amsterdoom's engine

Section 2.2 already described reflection and transparency. Transparency was either screen-door or (alpha) blended. While this is a great and useful technique, it does not handle refraction. A possible solution to this problem is somewhat like the solution for reflection, so lets review that solution first.

Reflection was achieved by mirroring the geometry in the reflective surface and rendering it from there. In essence, the reflective image corresponds to an inverted image from a viewpoint on the other side of the reflective surface. Whichever way is chosen (reflect geometry or reflect viewpoint), to make a distinction between reflective surface image and the rest, a *stencil buffer* is used to mask off areas of the screen. Additionally, this process can be repeated recursively when multiple reflective surfaces exist. Amsterdoom only supports a reflection depth of one, or only a single iteration.

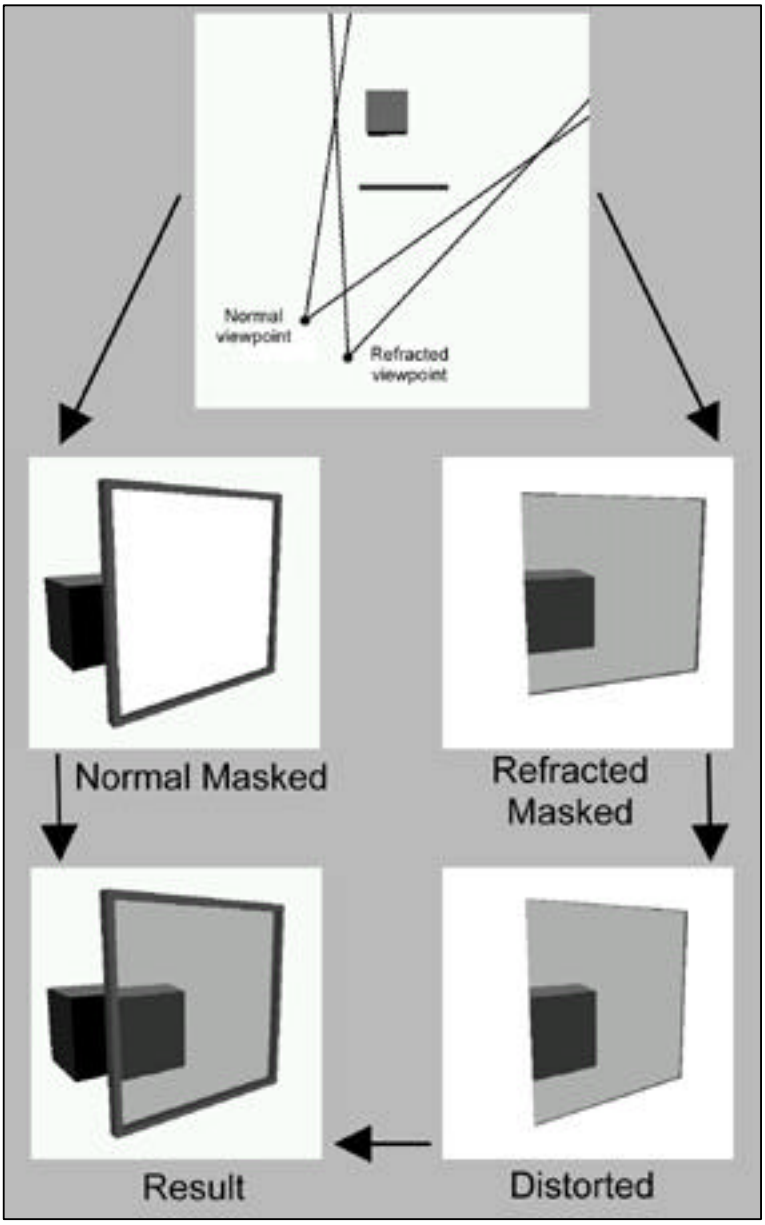


Figure 14 - Refraction

The same principle can be used to generate refractive transparency ([16], [18]), but it is a little more involved then with reflection. Whereas a mirrored image directly corresponds to the reflective surface to which it maps, a refracted image must be distorted to fit. We can achieve this by changing the viewpoint, but care has to be taken with objects that might now lie in front of the refractive surface, and have to be skipped in rendering. Also note, that this method only produces a close approximation to the refractive image, not true refraction.

Transparent surfaces do not have to be either completely opaque or complete transparent. They can be *translucent*, meaning one can still see the surface, but can also see through it. This is the kind of transparency one gets with screen-door and alpha blended transparency. Furthermore, a reflective or refractive surface can have a translucency value. When rendering the reflective or refractive image, the surface can be kept as a transparent surface (without refraction) and alpha blended with the scene behind it. This creates the illusion of a shiny surface (instead of a mirror) and translucent material.

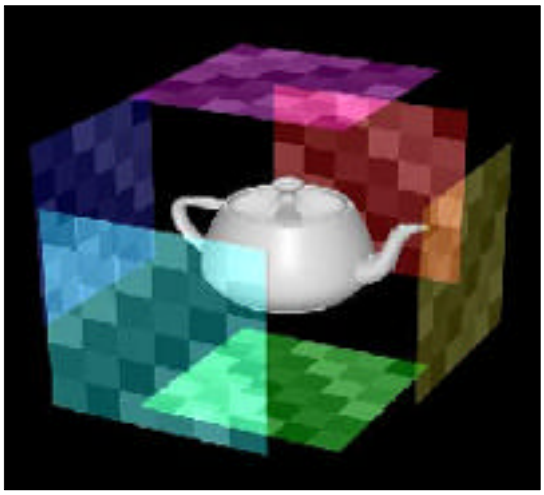
In addition, translucency can be simulated by fogging the objects seen in the reflected/refracted surface ([16], [18], [17], [53]). By having the objects be more clearly visible when they are closer to the surface, objects seems to disappear as they get further away. The speed at which this happens can be set per object. Note that fogging to black is used for the reflection and fogging to white is used for the refraction.

Another way to create reflections is by means of an *environment map*, also called a reflection map. This is a simple yet powerful method of generating approximations of reflections, specifically in curved surfaces. When a ray reflects of an object, instead of finding the intersection with the closest surface (as ray tracing does), environment mapping uses the direction of the reflection vector as an index to an image containing the environment.

To use environment mapping, first a two-dimensional image of the environment, the environment map, must be generated. Then for each pixel that contains a reflective object, the reflection vector is computed using

the view vector and the normal at the location on the surface of the object. This reflection vector is used to calculate an index into the environment map and its texel data is used to colour the pixel. Several projector functions exist that map the reflection vector into one or more textures. Two of these will be discussed next, namely *sphere mapping* and *cubic environment mapping*.

Sphere mapping is most commonly used for environment mapping in graphics accelerators. It is supported by OpenGL and indirectly in Direct3D. The environment texture is derived from the appearance of the environment as viewed orthographically in a perfectly reflective sphere. It is however rarely used due to severe limitations by the map shape. Sphere mapping is useful only for one orientation and one viewpoint, limiting its use to very specific conditions. Recalculating the environment map on the fly is too expensive for practical use. The technique by [33] solves this problem and shows some promise. However, there is another promising technique.



Figuur 7 - Cube environment mapping

Cubic environment mapping maps the environment onto the sides of a cube, with its centre corresponding to the centre of the environment. It is already widely being used in computer games, although not for reflections. As detailed by section 2.4 this type of environment map is commonly used to represent the sky (and other features outside of the geometry, such as skylines). They can be generated relatively easily and could even be generated on the fly (Unreal generates its skybox environment map dynamically). Cubic environment mapping is currently supported in DirectX® 7 and OpenGL®, and is implemented in nVidia's new GeForce 256™ GPU (Graphics Processing Unit) ([62]). Expectation is that because of this, more graphics hardware will support it.

Important uses of environment maps include specular reflection and refraction ([62], [52]). Furthermore, by blurring the texture the surface can be made to appear rougher. Other advantages include speed and versatility. However, generating and storing environment maps can be costly in time, effort, and resources. In addition, flat surfaces usually do not work well in conjunction with environment mapping ([53]).

Figuur 8 - GeForce 256 GPU

The natural inclusion of reflection and refraction is one of the strong points of ray tracing. By actually reflecting or refraction the light rays, extremely realistic effects can be realised. Still, currently environment maps are used to create reflections, specifically to increase performance. Computer games keep it rather simple. Reflections are simple planar reflections and transparency is achieved by either a screen-door technique or alpha blending. Transparency is used most often for water and highly transparent surfaces (glass and such) and reflection is used either for mirrors in bathrooms or to give mirror like properties to floors.

How would computer games benefit from reflection that is more advanced and refractive transparency? Glass windows usually do not have a very high refractive index and it is hardly noticeable if this index is zero. So current transparency is good enough for about half its application in games. Water however is highly refractive. Not only would it add to graphical realism to incorporate refractive transparent water, but it could also change gameplay. Imagine how difficult it is to hit someone in water, with that person not being exactly where somebody outside the water sees him or her.

Since water, and most other applications of transparency in games, consists of a flat surface, the secondary viewpoint method might very well work, provided that the problems with this method are taken into

consideration. Maybe only objects on the other side of the surface should be rendered for the refractive viewpoint, which should not be too difficult in BSP tree rendering. With the predicted increase of the use of curved surfaces, it is also important to look at transparency in relation to non-flat surfaces. Transparent curved surfaces look strange without refraction, yet the second viewpoint method does not work here. Using environment mapping for refraction might be the solution here.

Reflection, being used for either mirrors or floors, can be quite accurately done using current techniques. These techniques could even hold out if application of reflection spreads to more surfaces, glass and water also reflect, as well as tabletops and metal, since most of these surfaces are flat as well. The calculation cost might be too high though, especially with all the duplicated geometry it has to render. On top of that, it does not work for curved surfaces. Again, environment mapping might be the solution, but that depends on how well it can be applied to flat surfaces. Furthermore, pre-calculated environment maps do not include dynamic geometry (models and actors) and thus might need to be calculated on the fly, taking away some of its benefit.

Another problem reflection (and transparency) currently has is that objects do not fade into the distance, as is the case in the real world. Fogging solves this problem, but requires additional calculations. Pre-calculating this effect in the environment map takes away that problem, but still has the problem of either static environment maps or having to dynamically create them. Maybe it is possible to pre-calculate the environment map for the static environment and use this map in the run-time calculations for the dynamic objects.

In conclusion, reflection and refraction add enormously to realism and might even influence gameplay. Current techniques might generally be adequate, but reflection and transparency (with or without refraction) need to be applied more. Every surface should have a certain degree of reflectivity (probably often zero) and surfaces should be able to be both reflective and transparent, something not possible now. Curved surfaces should also include them and while the second viewpoint method is good enough for the moment, this probably calls for the increased use of environment mapping in the future. Possibly, they will coexist, but if environment maps are applicable to flat surfaces then there really seems to be no need for keeping the second viewpoint method.

4.2 Shadows

Where shading gives an object depth, shadows give a scene depth. Thus, shadows provide the user with visual cues about object placement. They are also important for realism. Like most games, Amsterdoom too supports shadows, but only two different kinds. Most shadows are pre-calculated into the light map, which gives problems with dynamic objects. Actors, especially the enemies, also have a shadow, but this is in the form a dark, blurry circle on the surface below them so they are grounded. Models (moving brushes) do not cast shadows at all.

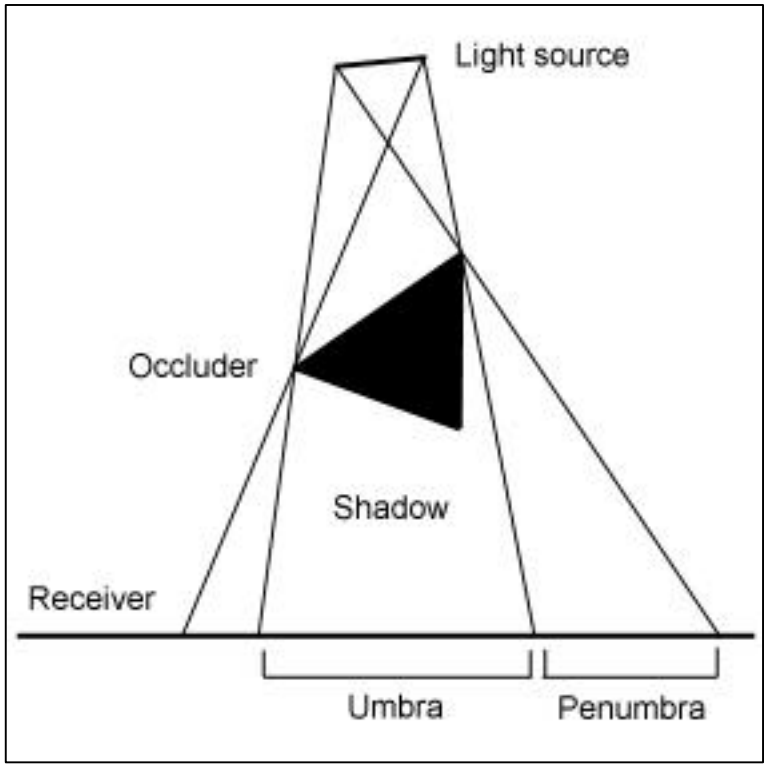


Figure 15 - Soft shadows terminology

In the real world, shadows are formed by the interaction between various objects. First, there must be a light source. Shadows are cast by *occluders* and the objects on which the shadows are cast are called *receivers*. Point light sources only create fully shadowed regions, sometimes called *hard shadows*. *Soft shadows* on the other hand are produced by area light sources. A fully shadowed region, the only type of region in hard shadows, is called the *umbra*. A partially shadowed region, mandatory in soft shadows, is called the *penumbra*. Soft shadows are generally preferable, since they are more easily recognised as shadows and hard-edged shadows can be misinterpreted as actual geometric features, but it is generally better to have a shadow than none at all.

The simplest kind of shadow is a blurred black circle applied as texture on the floor. While very simple and

very inaccurate, it can anchor a person to the ground and is thus preferable to no shadow. A method used to create hard shadows is by projection. In this scheme, a three-dimensional object is rendered a second time in order to create a shadows. In a sense, the object is projected on a receiver to create a shadow. Care has to be taken not to let the shadow stick out. For this, a stencil buffer can be used. This is simply a mask that lets you only draw in the designated area (the receiver). A common technique in games is to store the shadow in the alpha component of a texture, and then move and rotate a textured quadrilateral to follow the movements of the shadow-casting object. Another method is to use a simpler version of the occluder for the shadow projection.

The projection method can only create hard shadows on planar surfaces. Soft shadows can be approximated by generating various images and storing these in an *accumulation buffer* (see also section 5.2). With this buffer, a blurring effect can be easily generated. One way to create multiple varying images is by using a number of point lights for the surface light. In practise, this might be difficult because of memory constraints or other factors. Instead, the location of the receiving plane can be moved up and down. By averaging the projections cast upon it, good looking soft shadows can be created using only a few samples. If a single shadow projection is used to generate a texture of the shadow, the need to project and render the occluder multiple times is eliminated. A few problems with this are that shadows are always larger then the objects casting them, which is not always the case with area lights in the real world, and shadows can leak out from under an object.

The above methods can only cast shadows on planar (flat) surfaces. One idea is to make an image of an occluder from the point of view of the light and use it as a shadow texture. A drawback of this method is that one must identify which objects are occluders, and which are their receivers. A few methods exist that generate correct shadows without the need for such intervention.

Shadow volumes make clever use of the stencil buffer to cast shadows onto arbitrary objects. Imagine an infinite pyramid with its top at the light point and its edges going through the vertices of the triangular occluder. The part under the triangle forms a truncated infinite pyramid, which is called the shadow volume. When casting a ray into the scene, one can keep track of how many shadow volumes it enters and exits. When the ray hits the object to be displayed at that pixel, having entered more shadow volumes then exited states that the object is in shadow. To avoid the tediousness of doing this geometrically, a stencil buffer can be used to do the counting. Note that when the viewer is inside one or more shadow volumes, this number of volumes should also be counted as entered. Unfortunately, shadow volumes greatly increase the number of polygons to render.

Another method to create shadows on arbitrary surfaces is with the use of a *shadow map*. This shadow map is very similar to the light map. The scene is rendered two times, once using only ambient light and once using the whole lighting equation. The shadow map then determines which of the two is used for each pixel, the ambient render where there is shadow and the lighting render where there is not. The advantages of this are that general-purpose graphics hardware can be used to render arbitrary shadows, and that the complexity is linear. Disadvantages are that the scene must remain static, the quality of the shadows depends on the resolution (in pixels), and the algorithm is susceptible to aliasing problems.

Computer games usually pre-calculate the shadows in the light map. This allows for fast rendering of shadows and, in conjunction with shading algorithms, shadows always appear to be soft shadows. However, this also means that dynamic objects do not cast shadows. The only dynamic shadows possible, are those that are cast by dynamic lights off static objects, and only when using a dynamic light map. Dynamic objects are rarely able to cast shadows. Some dynamic objects pretend to cast shadows by placing a blurred black circle on the floor.

Figuur 9 - Blade

Things are starting to change in this field, however. A 3D-acceleration card exists that supports stencil buffers and games currently under development, state that they will support dynamic, actor cast, shadows. Examples of such games are Nihilistic Software's *Vampire, the Masquerade – Redemption* and Rebel Act Studios' *Blade*. Better yet, they are already supported by games such as Bungie Software's *Myth* and Appeal's *Outcast*. Furthermore, specialised patches that work only on certain graphics card or some specific conditions have been made to let games such as GT Interactive's *Unreal* and id Software's *Quake 2* include dynamic shadows.

On an interesting note, the engine for Id Software's *Quake 3 – Arena* supports dynamic shadows. Yet, they have been left out, because play testing revealed that the shadows, due to the strange lighting, cause players to

constantly mistake their own shadows for enemies. The author has no more information about this, so it is difficult to say if this will be a recurring trend in the future or if it was just an isolated incident. It is more likely though that the latter is the best.

Especially in *Vampire* and *Blade* (as seen in in-game preview movies), the dynamic shadows add a tremendous amount to atmosphere and realism. This might be due to the fact noted in Amsterdoom that contrasts in light look far better than realistic, but somewhat uniformly lighted areas. However, it is just as likely, if not more so, that this is due to the presence of shadows in reality. If one is in an area lighted by few light sources, one expects to see sharply contrasting shadows. In *Outcast* and *Myth* the effects are less apparent (from what little the author has tried of them), probably because they are both set mainly in large, outdoors, well-lit environments where one does not expect to see very evident shadows.

Shadows are the natural result of the blocking of light. Since lights are needed to see anything at all, it only follows that one of those lights will be blocked at some point. If this does not create shadows, it will look strange and unnatural. However, shadows do more than conveying a realistic image. They also add depth to a scene and thus help viewers ascertain spatial geometric relations between object. As such, shadows seem of integral importance to computer games.

Similarly, having some shadows is better than having none at all. Because of this, and because they are easier to implement, computer games will most likely start by supporting hard shadows. Likely, some blending or transparency technique will be used to convincingly merge them in the background and have them vary in strength. Then, maybe starting through some blurring first, true soft shadows will start to see the light of day in computer games. On an interesting note, 3dfx claims that its next generation 3d-acceleration cards will already support soft shadows through its T-Buffer technology, probably an advanced version of the accumulation buffer.

4.3 Indirect Lighting

When in Amsterdoom lights are only placed there where they are in the real world, one will soon notice that light and colour seems to be missing on certain surfaces. This is because Amsterdoom does not use *indirect lighting*. Indirect lighting is used to indicate that light does not only reach a surface directly from the light-source, but also by bouncing off other surfaces.

To include indirect lighting in ray tracing, a large number of rays have to be spawned to find out if any light reaches the surface via other surfaces. This is highly impractical and extremely costly, making this solution impossible. Ray tracing may be good at specular reflection and refractive transparency; it can not handle indirect lighting very well. Instead, it uses a directionless ambient lighting term to account for all other global lighting conditions. Luckily, there is a better method.

Radiosity systems are based on theories of heat transfer between surfaces. Surfaces are subdivided into a mesh of planar *patches* over which the radiosity is constant. This radiosity is the rate at which energy leaves a surface, which is equal to the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Unlike ray tracing, radiosity methods first determine all the light interactions in an environment in a view-independent way.

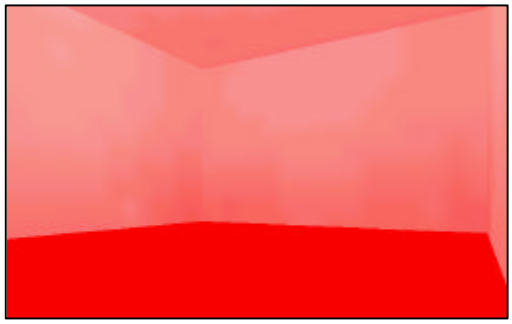


Figure 16 - Radiosity (white walls & red floor)

In radiosity methods, all surfaces can emit light. Thus, every light source is modelled as having area. The radiosity of a patch is computed from the radiosity values of all patches, considering visibility. Important to calculate the influence from another patch on a patch is the geometric relationship between the two patches, known as the *form factor*. The radiosity of the patches has to be recalculated several times, as the radiosity of the other patches keep changing. Executing this radiosity algorithm is costly and storage requirements for the result are high. Luckily, one can adjust the number of iterations used to calculate radiosity. More iterations generally give a better result, but cost more time to calculate.

Except for the high costs, radiosity has a few other problems. First, the view-independent nature of the radiosity computation usually precludes the support or specular reflection. Secondly, and more important to computer games, the environment is relatively static, except for view changes. This, because any object movement requires that the form factors are recomputed.

Yet, the fact that it can be pre-computed makes it possible to use it in computer games. Radiosity can be pre-computed into the textures or the light map. Doing this in the textures increases texture storage far too much, so doing it in the light map is generally the better option. After all, that is what it is for (keeping track of the pre-computer illumination of surfaces). Note that this still does not allow moving objects in the environment, or reflections. Nevertheless, for the latter there is a solution.

Radiosity is very good at producing soft shadows. It produces them easily and convincingly, adding much to atmosphere. Specifically, radiosity is very good at producing gentle colour gradients. However, sharp changes in colour can be a problem, especially with shadows. Furthermore, it cannot produce reflections on shiny surfaces. Some of these problems can be fixed by using a finer mesh, but that does not solve the reflection issue. Luckily, ray tracing can be applied after radiosity ([69], [77]). This has the benefits of both radiosity and ray tracing.

A number of 3D engines support radiosity, for example id Software's *Quake 2* engine ([12]) and Eclipse Entertainment's *Genesis3D* engine ([19]). Of the upcoming games, Remedy Entertainment's *Max Payne* ([72]), 3D Realms Entertainment's *Prey* ([50]) and ION Storm's *Daikatana* ([14]) are said to support radiosity. Not al computer games use the radiosity capabilities of their 3D engine, though. For instance, Amsterdoom does not use radiosity due to some problems with the radiosity solution in the engine.

With radiosity in computer games, the lighting is usually pre-calculated into the light map. This way it can still normally use its other effects like transparency and reflection. This is similar to using ray tracing after the radiosity solution. Moving objects is unfortunately still not possible. In computer games, they are usually left out of the radiosity calculations (although they do get radiosity values from the environment). During run time, they usually keep the value they received on their starting position, which might look strange. So, radiosity with respect to moving objects is handled the same way as shadows, they do not influence the environment but do get a value in their starting positions.

There are no real solutions to use radiosity in a dynamic environment. Nevertheless, it provides such an increase in realism that it is often worth these inconsistencies. A solution might be to give each moving object its own, separate (possibly interpolated) set of light maps, something similar to the synthetic lighting method proposed for photography in [27]. This is however just speculation on the part of the author, since no research has been done in that area by the author.

Radiosity, when used properly, adds as much to realism as reflection and refraction. Although it performs poorly in dynamic environments, even more so than ray tracing, it seems worth to use it. Especially combined with ray tracing based techniques it can produce stunning results. Radiosity is even more useful to computer games because it gets its stunning results from pre-calculating a lot. Of course, therein lie its problems, pre-calculation may be able to do a lot, but making dynamic adjustments is usually not possible or at least very demanding. Seen purely from realism, radiosity and ray tracing techniques should and probably will work together in the future, unless something better is found.

4.4 Lighting Conclusions

"Better to light a candle than to curse the darkness." This Chinese proverb not only states the importance of light, but also the importance of trying to implement it correctly. Even more, it also relays the thought that it is better to have some than none at all. This entire chapter has been devoted to ascertaining the importance of possible future aspects of lighting. This included looking at transparency, reflection and refraction, hard and soft shadows, and indirect lighting in the form of radiosity techniques.

To be able to make a game of any kind, lighting is important. More specifically, a clear difference in pixel colour must be apparent before any interaction can be included. Current computer games have come a long way since the old, essentially ambient, hand lighted graphics. Since environments have become more dynamically interactive, a distinction is being made between static objects and dynamic objects, and thus a distinction between the lighting of static and dynamic objects exists.

Especially the lighting of static objects has advanced a great deal. Where *Doom* had only section based lighting, static objects in current 3D computer games are now fully shaded and shadowed. Even dynamic objects are fully shaded, although for models this shading is generally not updated yet with change in the position. Of course, still many things could be improved on this point. The light map that is used can provide better quality shading than Gouraud shading alone, but still not nearly as good as Phong shading.

For dynamic objects, things look a little more dismal. Models are generally not shaded dynamically. Although *Unreal* does support, and often uses, change in shading with position, it is not automatically turned on for models, due to the added rendering cost and storage space. Actors are lighted separately, so they usually do respond to differing lighting conditions per location. The real problem lies with shadows. Models do not cast

shadows and actors can only pretend to. While in a fast paced game their absence might not be immediately apparent, including them would make a great difference as to the perceived realism.

Illumination is not the sole function of lighting. Light can reflect off objects and pass through some. This lets observers see things they normally would not see, in a way they normally would not see it. Current implementation of transparency is always straight, simple, non-refractive transparency by either alpha-blending or screen door methods. A reflective surface acts more as a portal to the same, reflected geometry (a parallel reflected universe would make a great game concept). Partial reflection can be obtained by combining reflection with transparency, but it seems that still much could be improved here.

Several of the more advanced lighting techniques have been discusses to improve on the situation described above. Section 2.3 in Chapter 2 explained shading and the different kinds of shading, which included the rarely used (in 3D computer games) Phong shading. Section 4.1 explained more advanced reflection and ways to include refraction in transparency. Section 4.2 explained different kinds of shadows (hard and soft) and different ways to create shadows. Section 4.3 described a technique, radiosity, used for the indirect lighting of a scene.

In about half the cases (or more) where transparency is used, refraction would not add a great deal to realism. Those cases would benefit far more from applying reflections to transparent surfaces (instead of using transparency as a way to make reflective surfaces seem less reflective). It would add a lot specifically in the case of water and, with the apparent future addition of curved surfaces, in the case of transparent, curved objects. Reflection is decently done, but only where a flat surface has near mirror-like properties. Transparency, but especially reflection, should be applied to far more surfaces, including curved ones. Each surface should have a degree of reflectivity, not to confuse with shininess. In addition, both would benefit from adding a fading effect to them, so objects further from the surface are less clear.

The technique that seems most useful to create refraction is very similar to that used to simulate reflection. Render the geometry behind the refractive surface with a stencil buffer, from a refracted viewpoint or with a refracted copy of the geometry. Of course, this technique only works with flat surfaces. The most promising technique for more advanced reflection seems to be the stencil buffer. This however, might not work with flat surfaces and has to be dynamically adjusted to dynamic objects. If these problems can be fixed, it is even a useful technique for refraction. Both reflection and transparency could use fog to create the fading effect.

Shadows add to realism even if they are not very realistic. Yet the more realistic the shadows, the more they add to realism. In addition, shadows help viewers in understanding the spatial relations between various geometric features. There is a difference between hard shadows, generated by point light sources, and soft shadows, generated by area light sources. Soft shadows appear the most realistic, probably since in the real world all light sources are area (or rather, volume) light sources. The most promising technique seems to be the use of shadow volumes in conjunction with the stencil buffer. Indeed, future computer games generate hard shadows in some way with a stencil buffer, although this is probably done by making an image of the occluder instead. The best way to approximate soft shadows appears to be by varying the location of the receiver plane and storing the various images in an accumulation buffer.

Radiosity is generally considered the best method for creating indirect lighting. It is less capable of generating reflection, but this is easily remedied by combining it with ray tracing techniques. However, a more serious problem is that it has a high computation cost. While this is of little concern to static scenes, because radiosity can easily be pre-calculated, it does pose a problem to scenes with dynamic elements. Leaving the dynamic objects out of the radiosity calculation gives at least the benefits of radiosity to the static objects, but does not solve the problem for the dynamic objects. There is no solution to this yet, although the author has speculated that it might be possible to keep the radiosity influences of various objects on the environment separate as to be able to calculate them separately.

Lighting and shadowing is very important to the realism of computer games. Various advanced techniques exist to create the most realistic effects, all with their various advantages, disadvantages and problems. The most problems in lighting come from the dynamics of objects and the viewpoint, so much so that they might need a separate way of handling lighting. Yet, due to the importance of lighting, it is especially significant that dynamic properties not be overlooked.

Chapter 5 – Future in Animation and Blurring

A well lighted, well-modelled world can still only deliver a single picture. Without animation, there is no ‘life’ to a computer game and thus not much fun. Thus it is important to make sure animations are realistic, since change is very noticeable to the human eye. Something else that is noticeable is when things are sharp when they should not be. Certain objects should be out of focus and objects in motion should not be too clearly visible. Even more important, the limited square grid of the screen (the pixels) forces sharpness on edges of objects in a way that it

should not. In those cases, it is better to have a little blurrier view, the human eye will correct for it. Thus, this chapter examines *animation* and *blurring*.

5.1 Animation

'To animate' is literally 'to give life to'. Thus, animation is movement that the observer can see. There are two types of animation; *computer-assisted animation* and *computer generated animation*. The first refers to computer techniques that aid the traditional, hand-drawn animation process which is not related to computer games, and hence this section will focus on computer generated animation. The techniques that will be described are *bone animation* (or *skeletal animation*), *weighted vertices*, and *physics*.

For animation, it is important to define the position exactly at the time of rendering. This is usually done by defining a few key-positions (or keyframes) and interpolating between them. While it is possible to handcraft each of these frame, in a sense similar to drawing frames in traditional, hand-drawn animation and indeed this type is used often currently, it is also possible to calculate appearance of the objects. The benefits of this are ease of animation and versatility, the latter in the sense that various animations can be easily combined to create one new animation sequence.

The most useful technique is by the use of *bones* that animate the object. This system of bones compares to the skeleton within humans. Now the object can be animated by moving and rotating the bones, who drag the vertices around them with them. There are various ways of doing this. Each vertex of the 3D object can be assigned to specific bones, or the vertices can be moved according to their proximity to the bones. Generally one bone influences more vertices and one vertex is influenced by more bones. Bones are linked in a hierarchical structure, the topmost bones being linked to a certain point that defines the position of the object, and each subsequent bone linked to the other end of the bone above it in the structure.

There are a number of ways in which to place the bones at the desired location. One is simply to rotate them by hand, starting with the topmost bones, called *forwards kinematics*. It gives ultimate control of the animation to the animator, but can be time-consuming in that every bone has to be rotated by hand. In *inverse kinematics*, one of the lower bones (often the lowest) is placed at the desired location. The rotation of the bones above it is then automatically calculated based on the initial angle between bones. Added benefit of this is that it is much easier to see certain animations (such as walking) as repositioning certain points on bones (such as the heel) instead of repositioning every bone. It can work the other way around as well, by moving a higher bone, which could leave points below it where they are.

Bone animation was already widely used in frame-based rendering, animation packages. In computer games, that system would be used to define the various keyframes by deforming the basic mesh and generating a mesh for each frame. However, bone animation systems are with increasing frequency used directly in 3D computer games. Now each object does not consist of a sequence of meshes, but of a single mesh and a sequence of skeletal positions. Amsterdoom too uses this approach.

With bone animations, various different animations can be blended into a single animation. This way an animation sequence can be made of a creature walking and waving from the animation of walking and the animation of waving, for instance. In addition, the bone animation system can be used to create complex facial expressions, as done in Shiny Entertainment's *Messiah*, for example. The bone animation system can also be combined with the levels of detail technique, so that animated faces only use a lot of polygons when they actually animate, again as done in *Messiah*.

The bone animation system defines the various poses for an actor. The surface of the actor has to respond in kind. Specifically in creatures and humans, the skin and clothing has to animate according to the change in posture, while remaining a flexible appearance. This means that when forces are applied to them at a point, they must be propagated through the object in some non-uniform manner.

A simple way to represent this is by modelling the object as a collection of mass points connected by springs. In polyhedral models (like actors), this translates to assuming that all vertices are points of equal mass and all edges are springs with the same spring constant. The initial edge lengths are used as the equilibrium lengths of the springs, leaving the user to specify only a single constant and mass. When force is applied to vertices, such as through collision, the force is at first concentrated at those vertices and then propagated throughout the object by the spring network.

A special case of this is the one in which the flexible material slides over another surfaces, such as in clothing and skin (depending on the specific representation of the model). The normal problems are exacerbated because of the near constant contact between the flexible material and the underlying support. With this method, skin can be stretched based on its location to come to a more realistic simulation. Muscles can be flexed, lips can be curled, and clothes can be rustled.

Figuur 10 - Vampire, the Masquerade - Redemption

This technology, also know as *weighted vertices* technology, was used in effects-driven movies such as Steven Spielberg's *Jurassic Park* and George Lucas' *Star Wars – The Phantom Menace*. Now it is finding its way to computer games. Nihilistic Software's *Vampire, the Masquerade – Redemption* is the first game to assign weight to each vertex in the character model. If the effect is good, which likely it is, the technique will probably be incorporated into more future games.

When asking for issues important to create realistic real-time 3D renderings, animation is often named. More specifically, *physics* is named as important to realism. By this, the natural response to gravity, acceleration, and collision in the motion of the object is usually meant. This further removes detailed control from the animator. Newton's Laws of Motion can be used to control an objects motion. Gravity, or acceleration following the gravity of earth, can be approximated by a constant acceleration.

Incorporating collision detection and collision response is one of the more realistic aspects of physically based modelling. This collision can alter the animation of objects. Because collision is tested once each time interval, collisions can be missed and will almost certainly not be detected on the exact time of collision. Other physics include the conservation of momentum and friction.

Most computer games already support most physics. Collision detection is used to make sure a player does not walk through walls, that things can be picked up, and that rockets actually hit an opponent. Gravity makes objects fall on the ground and friction makes sure objects actually come to a standstill if stopped accelerating. What is less noticeable in computer games, although often present, is conservation of momentum.

It is very difficult, although possible, to make a computer game without animation. Animation lets the user directly see the results of his actions, but more importantly, it gives life to the virtual world in which the user is to be immersed. Bone animation offers perfect tools for animators to animate objects, weighted vertices makes creatures and humans appear flexible, and physics makes sure animation occur according to natural laws. Because of this, animation is very important and the techniques described in this section will likely be incorporated in future games.

5.2 Blurring

When simply rendering a scene, everything looks very sharp. Far objects are as sharp as near objects, moving objects are as sharp as they would be still, and everything is sharp and blocky. These problems can be fixed by *depth of field*, *motion blur*, and *antialiasing* respectively. This section details the problems and describes solutions.

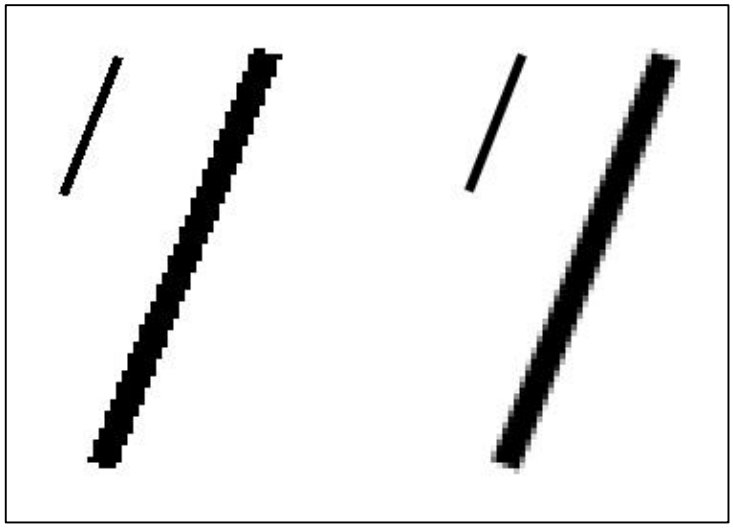


Figure 17 - Aliasing & Antialiasing

When rendering a scene, the pixel values must be determined that will most accurately depict the underlying scene. One could sample from the centre of each pixel, but that has a problem. Because of near hits and near misses along the edge of an object, the edges can look jagged. This problem is called *aliasing*, and efforts to battle this phenomenon are called *antialiasing* techniques.

Antialiasing is that for pixels that have multiple objects (including the background) visible in it, one should take a value averaging the values those individual objects would give. This makes edges a little blurry, which is generally better than

having a noticeably jagged edge. One natural idea is to take more samples per screen grid cell and blending these in some fashion. Such methods are called *supersampling* techniques. Higher sampling rate can be implemented in three ways. The simplest version is to render at a higher resolution and then combining neighbouring samples to create an image.

A related method is the use of an *accumulation buffer*. This buffer has the same resolution as the intended image and multiple images are generated and summed up in it. After rendering, the images is averaged and sent to the display. The additional costs of having to render the scene multiple times per frame and move the result to the screen makes this algorithm costly for real-time rendering systems.

The third method is the *A*-buffer, sometimes called *multisampling*. This is used commonly in software to create high-quality renderings, but at non-interactive speeds. Supersampling is used by calculating a polygon's approximate coverage of each grid cell. Polygons are sent to the *A*-buffer to create *coverage masks* or each screen grid cell it fully or partially covers. Pixel colour is computed by determining how much of the mask of each fragment is visible, and then multiplying this percentage by the colour of the fragment and summing the result. There are some problems with this method, like the limited size of the coverage mask, the use of a *box filter* (simply adding samples regardless of position), and the lack of gamma correction.

Another sampling-related antialiasing approach is to distribute the samples randomly over the pixel, called *stochastic sampling*. This works better because the randomisation tends to replace the aliasing effects by noise, to which the human perception is much more forgiving. The most common of these techniques is *jittering*. In this, the pixel is divided into regions of equal area and in each area, a random location to sample from is chosen. Unfortunately, stochastic sampling is not practicable in real-time systems.

There are other methods, such as edge and full-scene antialiasing, each with their own problems. In general, the *A*-buffer algorithm seems to be the most promising (see [90] on hardware implementations of this algorithm). Yet, the T-Buffer introduced by 3dfx seems to be more an advanced version of the *accumulation buffer*. Therefore, this algorithm should also not be passed by as a potential future real-time antialiasing technique.

In a movie, when objects move across the screen there is *motion blur*. This is a subjective process, if a camera is tracking an object, the object does not blur, but the background does. Motion blur is usually added to computer graphics for the same reason lens flares are added: to give a psychological cue to the user. To add to the feeling of motion, the blur may be overemphasised.

On of the ways to accomplish motion blur is by modelling and rendering the blur itself. In fact, this is why lines are drawn to represent particles (see section 2.4). This idea can be extended by adding polygons to objects. With the use of transparency (opaque where the polygon meets the object and transparent on the other end) polygons can be used to represent the direction of movement of the blur. An example of this would be adding such polygons to a sword slicing in front of the viewer.

Another way to achieve motion blur is by the use of the accumulation buffer. The object is moved to some set of the positions it occupies and it is rendered into the accumulation buffer. The result gives the blurred image. This process is counterproductive for real-time rendering. A clever way to use an accumulation buffer is by keeping it from frame to frame. Then render the current model each frame, but also the model a set number of frames back. This last rendering is subtracted, instead of added, from the accumulation buffer. This way, only two renders are needed each frame to create the blur effect.

Another application of the accumulation buffer is *depth of field*. Depth of field is the effect where objects at a certain range are in focus and objects outside this range appear blurry, the further outside the blurrier. With the accumulation buffer this effect can be simulated be varying the view slightly, while keeping the point of focus fixed. In this manner, objects will be rendered blurrier relative to their distance from this focal point.

The real world is not an in-focus, square grid, motionless affair, so real-time computer graphics should not be either. The techniques discussed above can help to achieve this. Interesting to note is that for all three techniques accumulation buffers can be used. Indeed, such a versatile technique does indeed seem to have a future, specifically since 3dfx' T-Buffer technology seems to be a more advanced version of it. Do not forget other techniques, though. The *A*-buffer seems to be a very good technique for real-time antialiasing and adding transparent polygons for motion blur produces some spectacular, yet simple, effects. Even depth of field might have other techniques, which require less rendering, to produce the effect. Until more hardware starts supporting the T-Buffer, it will only be a temporary phenomenon.

5.3 Animation and Blurring Conclusions

This chapter gave a few other issues that are important to graphical realism in computer games. Particularly it looked at various levels of animation and at various additional techniques which make an image blurrier, but more realistic.

Without movement there is only a still picture, without animation there is no interactivity. Section 5.1 researched various animation techniques with the intent to discover how they can be used to increase graphical realism. The techniques described for this were bone animation, weighted vertices, and physics.

Bone animation is used to define the various postures of animated figures (and objects). The benefit of it is that only a few lines (the bones) have to be moved to change the position of the entire mesh. Then there is the benefit of inverse kinematics over forward kinematics, needing even fewer changes. Bone animation can also merge several animations into one, which makes for more versatility with less animation sequences. Of course, bone animation does require the recalculation of the mesh with each change in posture, but despite this it seems that bone animation will become standard in the future.

Weighted vertices is a technique that works perfectly well with bone animation. It lets flexible material, such as clothing and skin, respond naturally to force applied to it. When force is applied to a vertex, it distributes it through the network of vertices. This technique too seems to have a future.

Physics on the other hand is already widely being used in computer games. It still has many imperfections though. This resulted in things like the “rocket jump” (shooting a rocket at the ground below to get extra momentum), “strafe running” (running forward by zigzagging for extra speed), and “jumpy behaviour” (jumping a lot to be a more difficult target). While these things would be impossible in the real world, they do add to gameplay (or so their practitioners claim). If this is true, then physics does not need to be improved in the future, but since they’re advanced playing techniques that is by no means certain.

To sum it up, bone animation and weighted vertices are likely to become standard techniques in the future (bone animation is already on the way of doing this). Physics might advance a little, but since current implementation is quite satisfactory, there will not be much change.

With the technique ray tracing gives, it is quite natural for computer to render a scene having everything fully sharp and in focus. This is not how the real world works. Light comes from many directions and the lenses of the human eyes have to bend those light rays to focus on a certain distance. Objects moving relative to the human eye leave images on the retina and thus cause motion blur. On the other hand, the screen is made up of a (very) limited amount of squares (pixels). Translating a continuous line to that can result in aliasing. In each of these cases the image has to be blurred somewhat to appear more realistic.

The idea of antialiasing is not to give a pixel the colour of the object beneath its centre, but to average the colour of every object under it (including background). This makes edges a little blurry, thus making the deficiencies of the pixel grid less obvious. The problem is to find all those objects. A few techniques have been presented and the *A*-buffer seems to be the most promising, since it can be implemented in hardware relatively effectively. However, it still has some problems and the accumulation buffer is another candidate.

Motion blur, like lens flares, is generally overemphasised to give a psychological cue to the viewer. There are two practical ways to generate motion blur in computer graphics. First is to place a possibly transparent polygon (or more than one) as the blur after the moving object. This is usually used for particles and the blur of swords and such. Another way is to keep track of various renderings and show a predefined amount of them. This can be effectively done with an accumulation buffer. Both techniques are viable in the future.

Human eyes focus on a particular point and the further another point is from this focal point, the blurrier it appears. This depth of field can be simulated in computer graphics with an accumulation buffer by rendering multiple, slightly varying, views of a scene. Other, potentially better, techniques may appear, but until that time this seems like the only viable way to achieve this. The question is, however, since computers have little knowledge of where a player is focussing his attention, if this does not ruin the experience (“Hey, I was looking at that. Why did that suddenly become ugly?”). Care has to be taken here.

Both antialiasing and motion blur have various techniques that seem to be very effective. The accumulation buffer, however, is something than can be applied to both of them and depth of field. It seems beneficial to use a technique that can be applied to so many things (including soft shadows, see section 4.2). It still has some problems, though. These techniques are not considered vital in computer graphics yet and it may be a while before the computer game graphics community makes any decision.

Animation is very important to computer games and it is noticeable if animation is not done right. Luckily, various techniques are surfacing to improve animated objects, specifically characters. Blurring is less important, although aliasing can significantly ruin a visual experience. At least techniques exist to deal with such issues, making the future look blurrier and more realistic.

Chapter 6 – The Future

So, will real-time ray tracing be the future of computer games? That question will finally be answered in this chapter. This will be done in section 6.1 by first reviewing how 3D computer games currently work. Then the advanced techniques described in Chapter 3 though Chapter 5 will be reviewed in section 6.2, as well as their relation to ray trace rendering, to come to a prediction for the near future. Finally, section 6.3 will look at the far future and will discuss what it is people want in 3D computer game graphics. The conclusion will be reached in section 6.4 at the end of this chapter.

6.1 The Present

To understand the future, one must first understand the present. This section refers back to Chapter 2 mainly. It describes which techniques are currently considered commonplace, which are considered state-of-the-art and which are just being developed.

Brushes, models, actors, and entities are used in nearly every 3D computer game these days, although the terminology might differ from product to product. Specifically first person shooters use them. Most computer games simply add brushes to the world to build the geometry, with the exception of games like *Unreal* that use CSG. BSP trees are also used by almost every 3D game. This is usually combined with some kind of portal technology. Within geometry, the use of curved surfaces is currently being developed (prime example: *Quake 3 – Arena*).

Texture mapping is also very common, and textures are often animated. Transparency has become common since *Quake 2* and, depending on hardware, is done through either screen-door techniques or alpha blending. Actors are nearly always skinned. Even more, the actors that represent the player have separate skins, so the player can easily create custom skins. Texture mapping commonly uses bilinear interpolation for magnification and mip-mapping for minification. The use of trilinear interpolation is considered state of the art, as is the use of more advanced forms of transparency and reflection. Anisotropic filtering might be a thing for the future.

In lighting, the most common types of light are ambient light and positional lights. Directional lights and surface lights are considered more advanced. Games generally use a light map, which can supports flat shading and Gouraud shading. There is no support for Phong shading yet, and it might take a while before there is. The same goes for specular lighting, although this technique might appear with the introduction of curved surfaces and hardware supported cubic environment maps. Shadows too are pre-calculated in the light map, in which way soft shadows can be simulated. Shadows only work with static geometry, however. Dynamic shadows, specifically for actors, are currently being worked on, but dynamic, soft shadows might be further away.

The different kinds of animation that are usually supported include camera motion, where generally the player controls the motion of the camera in some way. Other types of animation commonly used are object movement (mostly for models), texture animation, and vertex animation. This last type is usually done by cycling through a pre-set sequence of vertex positions. Skeletal animation is considered state-of-the-art and dynamic vertex manipulation is being worked on. Waypoint systems are starting to become more common, but such a system is just one type of artificial intelligence tool. Collision, gravity, and acceleration are usually included in the physics simulation, but better physics models might be introduced in the future.

One measure usually seen as a determination of how state-of-the-art products are is by the special effects they include. As such, almost all of them are considered state-of-the-art. Effects are becoming more common, though. What is already common is billboarding. In fact, while it is still used for skies, its use for just about everything else is slowly being replaced by actors and such. It might even be said that the less billboarding a game uses (while still looking great), the more state-of-the-art it is. Currently billboarding is used most commonly for things like fire, explosions, smoke, etc.

Every game is different and as such the way things are done, is different in every game. There are still some commonalities in many games, though. If a technique works, then why should games not use it? Only when two state-of-the-art games are being developed, and for each, the same new technique is being researched, will multiple possible answers arise. Then the particular implementation and hardware support will dictate which technique is used more often. In addition, some games will have different requirements, Meaning that something new must be found. The solutions described in this chapter may be standard solutions, but they are by no means solutions for every case.

6.2 The Near Future

The computer games industry is a very dynamic industry. With each new state-of-the-art computer game, new techniques are introduced. The techniques considered most significant have been discussed in this paper, to be able to ascertain whether real-time ray tracing will be the future of computer games. This section briefly reviews those techniques and specifically relates them to ray tracing. This not only to give an overview of which techniques are currently being worked on and which might be seen in the future, but also to see these developments in the light of ray tracing.

Curved surfaces are often considered *the* addition to real-time 3D computer graphics. Ray tracers and other offline rendering software have included them for quite some time and hence created geometry that is far more realistic. The 'inventors of the first person shooter genre', id Software, are the first to include it in their next game, *Quake 3 – Arena*. Because they have been innovators in real-time 3D engines it is more than likely that more computer games will soon include curved surfaces as well. Already there have been various indications in this direction, like the game *Messiah* that uses them. Note however that certain techniques, like reflection and refraction, are not available for real-time curved surfaces yet, and that computer games will, at least at first, only use Bézier patches.

The obvious benefit of levels of detail techniques is that scenes that are more complex can be displayed without having too profound an impact on rendering speeds. Additionally, the techniques make geometry scaleable to less powerful machines. These benefits make that it will be included in games such as *Team Fortress 2*, *Messiah*, and *Vampire, the Masquerade – Redemption*. Because the techniques are so much more powerful for computer games then they are for offline rendering, real-time graphics will incorporate them much faster then ray tracers, etc will. Thus, it can be expected that more games will use them in the future. That is, until rendering speeds are high enough to reach high enough amounts of detail with these tricks.

It will take a while before displacement mapping is included in computer games, if it ever truly is. Until that time, computer games have bump mapping to look forward to. Matrox have made it the key feature of their *Matrox G400* 3D graphics card. With that card, games like *Expendable* and *Descent 3* support bump mapping and games like *Outcast* support it on their own. More games will likely support it in the future, but while only one card offers it, support is not likely to be widespread.

The most important techniques ray tracing offers to create realistic 3D images, are those for reflection and refraction. A certain graphics cards support stencil buffers, which can be used with a secondary viewpoint to realise both reflection and refraction. For curved surfaces other techniques are needed. The technique most supported in hardware, as well as in OpenGL and Direct3D, is sphere mapping. This technique still has a few problems, which is why the new GeForce GPU supports another technique, namely cubic environment mapping. It does this together with DirectX 7 and OpenGL. Computer games have not gone further then secondary viewpoint reflection (by copying geometry instead of the viewpoint) and only have refraction-less transparency. However, with the support of these more advanced techniques and the upcoming curved surfaces, computer games are bound to see more reflection and refraction.

Shadows are used to accentuate the effect of light, and as such are important to realism. In fact, it is better to have inferior shadows than to have no shadows at all. Because of this, most 3D computer games support shadows in some form. Dynamic shadows pose more of a problem, and especially dynamic soft shadows. Dynamic hard shadows can be created with a stencil buffer, as supported by a certain 3D graphics card and as done in *Vampire, the Masquerade – Redemption* and *Blade*. Soft shadows can be achieved with an accumulation buffer. Such a buffer is a key feature of 3dfx' next generation 3d-acceleration cards, specifically the Voodoo 4 and Voodoo 5 cards, by the name of the T-Buffer. Because shadows are so important, an increasing amount of games will have increasingly advanced shadows.

Radiosity is a technique that creates indirect lighting and soft shadows. It can be added to ray tracing methods, as long as the geometry to which it applies is static. This because the technique is realised by pre-calculating, which is also why it can be applied to computer games. Current 3D engines, like the *Quake 2* engine and the *Genesis3D* engine, already support this and upcoming games such as *Max Payne*, *Prey* and *Daikatana* will support it. As long as 3D games gain rendering speed by pre-calculating will radiosity be a technique that can be successfully incorporated in future games. However, because radiosity is even less suited to real-time calculation than ray tracing is it will not be included real-time until computers are fast enough.

Most of the problems real-time 3D graphics pose are not directly due to animation, but because they need to be interactive. Animated geometry, specifically actors, can gain this interactive animation though the use of skeletal deformation techniques. Such techniques are to be employed in the games *Vampire, the masquerade – Redemption* and *Messiah*. A technique to naturally animate the surface (skin) is the weighted vertices technique, used in the movies *Jurassic Park* and *Star Wars – The Phantom Menace* and now in the game *Vampire, the Masquerade – Redemption*. Physics is already widely used in games, but will be more advanced in future games. All these animation techniques will be used with increasing frequency in future computer games.

Rendering techniques can create fully sharp images relatively easily, but the real world rarely is that clear. Depth of field is a technique that allows objects a certain distance from a focal point to appear blurrier and motion blur makes objects in motion blurry. While not used in computer games yet, the techniques are supported by 3dfx' T-Buffer. This T-Buffer also provides full-screen antialiasing, so it seems to be a very powerful tool. Still, it is only supported on a single graphics card and must first prove itself before it will be widely used.

For the future of real-time 3D computer game graphics, the techniques described above are not the only things of influence. With the highly competitive nature of the graphics hardware market, advancements are continuously made and are just as important. These advancements are made because companies implement particular 3D functions in hardware to differentiate their product from the others. Such functions include bump mapping (see section 3.3), stencil buffers (see mainly sections 4.1 and 4.2), and 3dfx' T-Buffer (see mainly section 5.2, but also section 4.2).

Another of these techniques is transform and lighting (T&L). According to nVidia, the first company to incorporate it in their graphics hardware, the technique is a 3D graphics breakthrough that enables a new class of applications ([63]). Indeed, it might be the next step in graphics hardware. Where until now 3D graphics boards only did rendering, and triangle setup and clipping, the graphics processor units (GPUs, as hardware using T&L has been dubbed) include the transform and lighting steps of the rendering pipeline. Previously, these had to be done by general-purpose microprocessors (such as the CPUs in PCs). This means higher graphics performance, since the specialised T&L engines process those functions faster than leading CPUs. It also means that the CPU power is free to do other things, such as physics calculations, levels of detail, artificial intelligence, etc.

A recent test between a GeForce card and the Voodoo3500 ([67]) has shown that the GeForce is indeed faster (in the higher resolutions) than its most direct competitor. With all the talk about the power of T&L, the differences are relatively slim though. This is simply because no game support T&L yet. Once they do, the difference will likely be very noticeable. A test (provided by nVidia) showed that the GeForce card could render 35,000 polygons at about twenty times the framerate (42fps) of the Voodoo3500. Moreover, there will certainly be games to support transform and lighting. *Quake 3 – Arena*, *Black & White*, *Messiah*, *Giants – Citizen Kabuto*, *Ultima Online 2*, *Testdrive 6*, *Vampire, the Masquerade – Redemption*, *Star Trek Voyager – Elite Forces*, and *Drakan 2* are but a few of the titles to support T&L. So the future of this next step in real-time 3D graphics seems to be secured.

Innovations will continue to be made both on the software side and on the hardware side. Question will be which innovations will remain. To be truly successful, hardware and software have to support each other. Hardware can offer great techniques, but if nobody uses them enough, they are of little use. The same goes for software; the techniques that receive enough hardware support will be more popular than techniques that do not.

6.3 The Far Future

The previous section showed that future computer games will incorporate more and more techniques to approach the graphical realism offered by ray tracing and radiosity. However, real real-time ray tracing is still a long way ahead. This section tries to reason where real-time rendering technology might go in the future, particularly whether it will ever reach real-time ray tracing. It will also try to estimate when ray tracing quality images will be reached in real-time rendering, which does not necessarily mean that real-time ray tracing is possible.

To understand where 3D computer games will be going in the future, it is important to understand what they are trying to achieve. Section 1.2 described the goal of computer games as creating an entertaining experience. It used immersion as a guide to measure how successful this attempt is. As part of what defines immersion, it stated realism and consequently, since this paper concentrates on graphics, the rest of this paper has been focussed on trying to create graphically realistic images.

This is not to say that unrealistic or even abstract computer game graphics cannot be immersive. In fact, *Tetris* with objects that look and fall realistically would not be much fun. It merely states that certain games that try to create a virtual world of some kind can benefit from adhering to a certain reality. This reality does not have to be the real world reality, but certain physical law, specifically those for light, should be adhered to. This is because those are the rules we are accustomed to and any deviation will look weird and distracts from the immersive experience.

It does not matter if a character can jump ten times too high, makes a "poing" sound when he runs into a wall and lives in a world so colourful it would make abstract painters jealous. Those are just part of the reality the game creates. However, it does matter if the character does not cast a shadow, looks crude and square (unless the character is a robot) and walks as if he should move five times faster than he is (feet gliding over the floor). Those things are just too natural for people to do without and still call something realistic.

Therefore, certain types of computer games, specifically 3D computer games, need to aim for higher and higher levels of possible realism. This paper discussed various techniques that do so, and can be added to the current technology. As time continues, more techniques will become possible and more things can be achieved. It is likely that Phong shading, specular reflection, displacement mapping, dynamic soft shadows, dynamic indirect lighting, and motion blur and depth of field will become possible. Additionally, the number of polygons real-time rendering engines can handle will increase; allowing for environments and objects that are more detailed.

Does this mean that real-time ray tracing will one day become the future? One is inclined to say yes, since it is just a matter of rendering power. Rendering power is something that has increased significantly, and is still doing so, over the past time. Moore's Law gives an acceleration rate of two every one-and-a-half year or about ten times every five years. Even better, game graphics chipsets are doubling in speed every six months. So it seems very possible indeed that real-time ray tracing might be available somewhere in the far future.

Several additional aspects need to be considered here though. First, 3D computer game graphics use tricks to come closer to the visual quality offered by ray tracing. They, in all likelihood, will not suddenly stop using these tricks in favour of ray tracing. Even more, these tricks will probably allow an approximation of the ray tracing quality to be available to 3D computer game graphics long before the rendering power needed for real-time ray tracing is reached. This will not make it likely that the additional speed offered by using these tricks is suddenly dropped in favour of the slightly more accurate ray tracing techniques.

Additionally, ray tracing is not the ultimate solution. It is very good at direct lighting, but can hardly handle indirect lighting. Also, it has some difficulty with naturally handling area lights (to create soft shadows). This is due to the fact that ray tracing tries to model lighting backwards, it starts with the camera and traces back to the light source. Indirect lighting and soft shadows are far better handled by radiosity methods. Thus, if ray tracing will ever be done real-time, it is even better to combine it with radiosity. That way one has the best of both worlds.

It is interesting to try to predict when personal computers have the rendering capabilities to achieve the same quality as offline rendering. Such a prediction is made by [53] by looking at Pixar's animated film *A Bug's Life*. It calculates that, to reach a real-time rate of 12 frames per second, a speed-up of about 150,000 times is needed. According to Moore's Law, this speed-up would be reached somewhere around the year 2024. However, according to the currently observed doubling of graphics chipset speeds, this could also be as soon as the year 2007. Of course, Bug's Life does not use ray tracing but a scan-line algorithm, but is a useful indication of when real-time ray tracing will be possible.

That calculation does not consider one thing, namely the use of tricks as described in this paper to achieve higher quality images. This makes it probable, indeed very likely, that the quality of images associated with ray tracing will more or less be achieved far before real-time ray tracing becomes possible. In fact, one might estimate that it is as little as three years away. The new *Playstation 2* console is thought by some to already be able to do *Toy Story* quality graphics. This is of course a little premature, but does lend credibility to the estimation made above.

The only benefit of ray tracing, over using trick, not mentioned yet is ease of programming. Ray tracing is a very intuitive and simple way to calculate the behaviour of direct light in an environment. It is also probably easy to put into hardware and is very suitable to parallel approaches, even down to the pixel level. Dedicated hardware already exists to aid offline ray tracing, in which multiple chips work in parallel. When ease of programming (which can speed up the development process) exceeds the need to create more impressive visuals, or at least when real-time ray tracing approaches can create graphics on par with other real-time rendering methods, will ray tracing find a place in both real-time 3D entertainment software and hardware.

6.4 Conclusion

This paper opened with the question: "Will ray tracing be the future of computer games?" Actually, the question should be more specified to: "Will ray tracing be the main technique in future of 3D computer game graphics?" To answer this question the introduction proposed to first look at which techniques from ray tracing will be incorporated in computer games, which techniques computer games will incorporate from other methods and what the future of computer games will be. These questions are answered in this section.

The techniques that future 3D computer game graphics will incorporate are varied. Reflection and refraction, the essential techniques from ray tracing, will probably be included. This goes for both flat and curved surfaces. Another sure bet is shadows, specifically hard shadows, at least at first. Curved surfaces (insofar as it is a ray tracing technique) will also be included. Various animation techniques that are not already included will also be.

Of the non-ray tracing techniques, radiosity is the most important to be included, insofar that it has not already. This, because it can so naturally handle indirect lighting and soft shadows. Bump mapping, and later maybe displacement mapping, might also be included in the future. That is if more graphics cards start to support it. The levels of detail techniques are more a trick than any of the others and will be included if only to crank up

the maximum amount of polygons actors can have. It would be better though if they were applied to all the geometry. Motion blur and depth of field would be nice to include, but might not for quite some time. It is important to remember that all these techniques can also be used in conjunction with ray tracing.

In the future, 3D computer game graphics will be able to approach reality increasingly. Specifically in the interaction of light with the environment, large advancements will be made. This will allow game designers to create the reality for their game the way they want, while still conveying a sense of realism to the players. It might take 24 years to get there, but eventually real-time 3D graphics will acquire the same level of realism as offline rendering currently has.

Since it is only a matter of computation power, it is, in the author's opinion, very likely that real-time ray tracing will be possible in the distant future, even though some are considering it science fiction. If only for aesthetic reasons, but more likely for marketing reasons, it is also very likely that real-time ray tracing will be used for computer games. It will take a long while though. At first real-time rendering will be done by using various tricks. Even when ray tracing is possible, the quality offered by using tricks, which is likely to be higher than that offered by ray tracing, will pass ray tracing by. Ray tracing is more likely to be used in conjunction with tricks that make up for its flaws, like radiosity and levels of detail. No matter how one looks at it, real-time 3D graphics will increase in beauty and entertaining value. Coupled with great gameplay, story, music, etc, there will be great game waiting to be played in the future.

Appendices

Appendix 1 – Wheel of Time Forum

During the development of the computer game "Wheel of Time" (see 1.1) the author of this paper followed the process closely by communicating with the designers and (other) players through the online "Wheel of Time Forum" (hosted by GT Interactive). At a certain point, the author of this paper posted a few questions pertaining to this paper. The original post and the answers are listed below. The author of this paper uses the Internet pseudonym "Overlord".

A note on the member status:

- "Member" indicates a registered member who has posted 31 or more messages.
- "Junior Member" indicates a registered member who has posted less than 31 messages.
- "Administrator" indicates an administrator of the forum (both appearing below are part of the Legend team).
- "Legend" indicates a Legend employee (which is the company making Wheel of Time).
- "Composer/Sound Designer, WoT" indicates the music/sound designer of the Wheel of Time game.

Author	**Topic: Graphical realism**
Overlord **Member**	*posted 07-27-99 09:26 AM* I would like to apologise in advance for the length of this post and for taking up so much of your valuable time. As a student I am currently working on a graduation project entitled "Graphical Realism in Computer Games". With "Wheel of Time" being one of the more advanced products being developed and with the close contact with the developers we have through this forum, I decided to post this message. What I want to know consist of several parts... First of all, what graphical aspects make a game (visually) more realistic? A list of possible aspects (including a few which I believe to contribute little/much) to consider: static and dynamic light, shading and shadows, animation, perspective, colour, curved surfaces, detail, transparency, reflection and 'special effects' like volumetric fog, lens-flares, particles and decals. In various previews and interviews it is stated that the game appears very realistic. What are the properties that achieve this? Some mention has been made of special effects like weather and scorchmarks, but do these really improve realism? Secondly, how are these aspects realised? Although I would specifically like to know how things are implemented, I can very well understand you not wanting to divulge such information. So I'm more looking for how are these aspects used to increase the sense of realism? And, maybe more importantly, why were certain things, which could increase realism, not included? As an example: In every screenshot I have seen so far the characters seem to be floating. This is due to the fact that they have no shadows. Although I am sure that in the final release they will have shadows, it serves perfectly as an example of what I mean: If shadows aren't included then why not, as they would increase realism to such a great extend. And if they are included, are they bitmap based or do you use real-time shadow calculations, and why. And finally, what would you do if machines were 1000 times faster? This is to find out what the future will bring. Would real-time raytracing be the way to do things, as is so often suspected? Maybe there are things nobody has fully considered before, due to limitations or other factors. And, more specifically to "Wheel of Time", what would you like to include in a possible second or even third "Wheel of Time" game to improve realism? Of course I realise that things like sound and interaction also contribute greatly to realism, but my project concentrates on the graphical side of realism. I also like to note that while these questions are geared towards "Wheel of Time" they also pertain to other (3D) games. And while I am specifically interested in the opinions of the Legend team, I would also greatly appreciate the opinions of others frequenting this forum. In any case, whoever you are, feel free to answer these questions as complete as you like. I thank you in advance. ---------- Game development is the Renaissance art of the information age. - Ben Sawyer -= Overlord =-
Funk **Junior Member**	posted 07-27-99 10:20 AM My personal take is that physics make up a great deal of realism. If you have a totally unbelievable object that moves like it does in the real world I am much more convinced of its realism than if you have a perfectly textured and lighted object that behaves oddly. One thing that is required for good looking, realistic physics is a high framerate. If you can combine this with realistic lighting and texturing then you have a truly killer feel.
Slava S. **Member**	posted 07-27-99 10:36 AM While I would leave the WoT questions for Legend to answer, I myself am an aspiring game developer, and would like to post some thought on gaming realism. The game I play the most right now is [still after 1/2 a year] Thief: The Dark Project. There is something about that game that haven't felt in any other game, this sense of immense immersion in to the world that makes you feel that your are really there (well, kinda). I tried to find out what makes this game so realistic and here are my findings. 1) I won't talk about sound and gameplay (which are easily the most important factors in Thief's level of realism) because that's not what this thread is about. 2) The "Level" Size- Thief's levels are huge (in comparison to games like Quake II and the like) and you can wander around for hours and you still wouldn't see everything which is pretty reminiscent of real-life cities and mansions. In Asheron's Call (which I'm beta testing) the world is one Giant level and you can run for hours and hours without getting to the end of it. 3) Similar Palettes - Just take a look at the Shadar Logoth screen shots and you will see that all buildings look blue, in fact everything looks blue. This is the same as in Thief where the whole mansion will have a similar colour theme to

	create an illusion that you are in the same place all the time. 4) Weather Effects and Day/Night shifts add a lot to the "atmosphere" in a game. Although absent in Thief, there are plenty of those pretty effects in Asheron's Call, and they add a lot to the realism of the game. I wish I could write more but I have to go and do work... ☹ Later, 3) --- Slava S. WoT Centre (http://wot.gamestats.com)
Speewa **Junior Member**	posted 07-27-99 11:06 AM It is only a waste of time if nobody gives feedback☺! You said some of the magic words. Lens flare. Can't stand 'em most of the time. I can't talk for everybody, but I for one do not walk around, or drive, or fly, with a 50-120 Zoom lens fitted over each eye. There are only 3 cases where a lens flare is appropriate that I can think of ATM, in a missile view (video from a CAMERA), a view when using some sort of periscope/telescope aiming device, and when you have action replays from track-side or following cameras. I am sure there must be some others, but the point I am trying to make is: Don't have a flare just for the eye candy. It needs to be in a proper context. This is the same for all graphics. You can have all the best effects in the world, but if you don't use them right, they won't help you. How many times have you played a brand new game that looks **great**, but then you go back to something with better gameplay, even though it is a year behind the times in the graphics area. Overall, if you have great gameplay the imagination of the user can 'gloss' over a few poor areas and fill in the gaps. When everything is good, you have a hit. If things where 1000 times faster, well SGI workstations with the Cobalt graphics chipset can use a live video feed as a texture right now, so I cant even dream what they could do with that much grunt. Some way to fade textures naturally into each other where they meet would be a good next step. [This message has been edited by Speewa (edited 07-27-99).]
Sam Brown **Administrator**	posted 07-27-99 11:52 AM This is going to sound really... bizarre... probably, but I'll do a bit of musing here and you can take it however you want. Realism or reality is in the mind of the observer. The more "graphically realistic" a game seems, the less graphically unrealistic gaps your mind has to fill. For example, if I look back to some of the more ancient (by today's standards) first person games, I find myself having to pretend that flat shaded untextured wall over there really has a metal look to it. On the other hand, looking at something like WoT, I don't have to flex my brain much to imagine that wall that has a light mapped brick texture really is a brick wall. Basically, the more realistic something appears, the less the observer has to work to believe it. So can we say that the measure of how realistic something is is actually a measure of how much work the observer must put into believing a fictional scene is real? Possibly... The real issue is to determine what is trying to be achieved. Is a game trying to visually simulate something that could occur in real life? Or, is a game trying to simulate reality...is it trying to feel real. These are, in fact, very different issues. A game that looks real is no more different than a painting that looks real, or a sculpture that looks real. It visually mimics what we know to exist in real life. So if I saw a screen shot from the Wheel of Time and thought, "wow, that looks pretty realistic" I am really thinking that it does a pretty good job of mimicking a real setting, but that's it, nothing more. The textures, the lighting, the fog effect, they all combine to make a pretty realistic looking scene. Am I fooled into thinking it's real? Of course not, I know it's a game, and in the end, the effects just make it look cool, they don't make it look real. Why don't they look real? As I said in the beginning, the more effects you add, the more textures, the more detail, the more polygons, the more lighting, the more fog, etc. the less you as the observer have to work to believe it is a realistic scene. The key is, there's still a gap, there's still the issue that you know what you are looking at isn't real. It may be the way the clouds move, the fact that the grass isn't simulated down to every blade, you don't feel the breeze on your skin, you can't smell the air, you can't taste the water... "So what?" you say, "it LOOKS real!", but I ask you, what is graphical realism if you know it isn't real? --- --=[Sam Brown]=-- Legend Entertainment sbrown@legendent.com
Glen Dahlgren **Administrator**	posted 07-27-99 01:04 PM Here's just a couple of cents: Realism isn't always the goal. When I started this game, I wanted beauty over realism. I didn't want the player to think he had stepped into a photograph; I wanted the player to feel he was inside a piece of fantasy fine art. You may attach the term 'realistic' to our game because we've done a lot of ground work to make our environments feel realistic--notably creating areas that have a solid architectural basis. But the textures, creatures, and effects have nothing to do with reality; they are about fantasy and fun. If you had a totally realistic game, you'd have an architectural walk-through game (which does exist, by the way, but I don't think it's all that great). BTW, unless Unreal suddenly starts supporting it, I don't believe that we'll include shadows for in-game characters. Sorry. --- Glen Dahlgren Legend Entertainment Company
Kaliv **Member**	posted 07-27-99 01:44 PM I would have to go along with Glen. Realism isn't always the goal...the 'beauty' is. Also, if you make graphics so realistic that it cuts down on gameplay, your game is worthless. I know you wanted more about graphics and such compared to realism...but as everyone knows, graphics are just a piece of the pie to make realism. But if you think of it, most games aren't all that realistic. For example, you get shot on the chest with a shotgun and you don't die. I'm not sure, but in real life, I think I would die. But more towards graphics...the blood from that wound would probably splatter all over the place...and drip on the floor after you. However, it just leaves a little blood decal on the floor where you were hit and a little blood goes flying. Yes, the blood is red, and some went flying and it interacted with the floor, but real?? Hmm... Nah I don't think

	it looks real enough in most games. Definitely not in Quake. But fun and neat looking...yes, that is where the beauty is. So I wouldn't say that all the advances for reality are necessarily for reality, but for the fun of the game. Shadows and dynamics lighting and fog you can barley see through...it puts a challenge in the game for better gameplay. Do I really care if it looks real?? No, not really...I want to play something fun and challenging with new effects. However, adding more things to make it look more real, sometimes makes the game more fun. Where would I like to see things goes if computers were 1000 times faster...virtual reality suits where the skill of the person makes how good a player you are...not just the skill of hand-eye co-ordination. Sorry, next time I write something long, I'll try to utilise paragraphs. [This message has been edited by Kaliv (edited 07-27-99).]
BenPatient **Member**	posted 07-27-99 05:26 PM Glen--I thought I should bring this up, cause maybe it will jog someone's memory. There was a card that came out--or was coming out--or something like that--anyway...it had support for shadows in unreal. I don't remember the details, but it was cool. They used realistic lightsource-based rendering for the shadows. For example: if you are going down a staircase and a light is on the wall as you run by, your shadow will shrink or grow in relation to the light. If you are in a baseball stadium with only two grandstand lights on, you will have two light shadows and one spot that is darker where they converge. If you are in a pitch black room in a team game and your buddy shines a flashlight on you, he would see a GIANT shadow of you on the wall and ceiling. I saw shots of all of these types of effects on a page somewhere, and I cannot for the life of me remember where it was. If anyone has the faintest memory of this, please say so, so I don't look like a goober ☺ Sorry if this isn't about realism--although the above effects would definitely help in the visual side of realism. Ben
Andy Frazier **Composer/Sound Designer, WoT**	posted 07-27-99 05:34 PM So...you could say the goal here is to go *beyond* realism...but not necessarily toward hyper-realism...more toward realism enhanced for the area. Does a tree seem more real in a forest than on a city sidewalk? Maybe, maybe not, but it looks more at home. What if it had blue leaves? Then your perception of how real it seemed would probably depend on the colour of the surrounding foliage. So if you're in a world where you can call on magic to defend yourself (some flavours of which defy certain physical laws...how would Newton feel about fire shooting from a piece of stone?), and you face non-human enemies that would seem decidedly out of place at your local wildlife preservation area, how appropriate would it be for the surroundings to look simply "real?" To me, it seems that would almost detract from the experience. I think, in that situation, you may expect the surroundings to look nearly as beyond basic reality as the power you control and the enemies you face. But...then you have to consider gameplay. If the world was *too* far out, it may take a little too long to learn how to interact with it effectively. So, you strike a balance. Say, for example, by using fantasy art as your model ;)...fantasy art that is based around earthlike structures & physics, but with a little something extra...a little something beyond. --- -Andy Frazier members.aol.com/mx4px
Petter Sundlöf **Member**	posted 07-27-99 07:52 PM Wow... lengthy responses... Realism isn't always necessary to create immersion, but it can definitely help. If the game is supposed to take place in an environment of which we already have a concept of how it looks, then a "realistic" representation helps with giving you a feeling of presence. Look at MDK, in my opinion a very good game which made me feel really involved, but did certainly not focus on realism - it showed me a new beautiful fantasy world. Although Overlord chooses not to speak about sound, I feel it is very important, and adds a sense of substance for the graphics. Looking at a door swing open, rocket flying by or something similar without a good sound pretty much ruins the illusion of reality, as achieving an idea of "reality" is a good combination of different impressions. Just my useless thoughts. --- // Petter Sundlöf (feldar@findus.dhs.org) Member of Shen an Calhar, The Band of the Red Hand www.calhar.dhs.org [This message has been edited by Petter Sundlöf (edited 07-27-99).]
Petter Sundlöf **Member**	posted 07-27-99 08:26 PM BenPatient: That was the shadow patch Creative released for their TNT card. It utilises something called a stencil buffer (don't ask me what it is ☺, that enables to draw volumetric shadows, relative to lightning (position, strength), object size, position et cetera. I don't think it was merged into the unreal codebase, but it would be neat if Legend just put it in (without adding it to the menus), maybe with a switch like "vol_shadows (n)" ☺The only bad thing is that you were only supposed to use it on Creative Labs TNT cards, but it worked with other TNT-based cards, you just needed a serial. --- // Petter Sundlöf (feldar@findus.dhs.org) Member of Shen an Calhar, The Band of the Red Hand www.calhar.dhs.org
Waffleboy **Junior Member**	posted 07-27-99 08:27 PM I think Glen hit the nail on the head earlier in this thread when he said that you're not always aiming for "realism". In this case, the creator is trying to make a stylised version of reality. How can one say if the balefire in the game looks realistic or not? All we have to go on is a description in a book. Speaking from a more technical standpoint now, and returning to the original question of what makes a game look realistic, I have a simple answer. I once had the opportunity to chat with Frank Delise, creator of the popular 3D-animation program 3D Studio Max and creator of many of the special effects used in the movie Lost in Space. In my experience as a computer animator (albeit amateur) I have found one of his statements during the chat to be very true: Lighting is everything. Crappy textures and mediocre models can be improved vastly by realistic and well-implemented lighting. And when I say lighting, this includes shadows. Form shadows give an object depth, and cast shadows give a scene depth. As for what I would do with computers 1000 times faster, well aside from creating worlds so realistic you'd need to hook your brain directly into it, I'd have to return to one of my previous points. Right now games are using mainly polygonal models, in fact the only game I know to use curved surfaces so far is Quake 3. If I had my way every organic

	model would be composed entirely of curved surfaces, and thusly the form shadows on such an object would be more realistic than current models, which use flat shading (i.e. grab the lightness value of the middle point of a surface, and make all the pixels on that poly that value).
radhermes **Junior Member**	posted 07-27-99 08:36 PM Overlord, Gotta little on your mind? REALISM is the key word. It sounds from Overlord's post that he understands the goal is not realism but play enhancement. All of the posters have touched on the critical point around gameplay and realism. If I may be so bold to restate the question. What decisions or priorities regarding graphical details enhance the player's immersion into gameplay? Obviously this question changes dramatically within the confines of the game. Because the type of RPG, FPS, etc, fundamentally dictates the ground rules of this question. Maybe this isn't what Overlord means at all.
Matthias Worch **Legend**	Posted 07-28-99 08:41 AM To add a really short answer from a pure LD perspective: I'm not trying to create 'realistic' levels. I'm trying to make CONVINCING levels (which don't necessarily have to exclude realism).
Taliesin **Legend**	posted 07-28-99 09:40 PM > Secondly, how are these aspects realised? I'm not at liberty to divulge implementation details at this time, however I will try to expand upon their uses in creating 'realistic' environments. Decals: The thing I like most about decals is that they add a sense of persistence to the levels. Not only persistence, but a form of character to the levels each time you play them. A quick example is in a multi-player game when you have a concentrated fight in one area of the level, that area will be stained with the blood and artefacts of the battle. Later when you revisit that area, you will be able to look back and remember what happened there. The next time you play the game, it may happen differently with some specifically intense battle happening elsewhere in the level. It'll never be the same twice. Rain: The first time I played Doom, and looked out one of the windows into the skybox, the first thing I wanted to do was to get out there and explore the area. We can (and do) do that now. In the case of rain, this is something you often see looking though windows, and wish you could go out and play in it. What we have done is brought the rain inside thus further connecting the player with the world around them and at the same time allowing them to literally play in it. (Note for the sane: The areas that the rain is indoors are where the roof has caved in, etc. It doesn't really rain indoors... we haven't completely lost it.) ---------- Aaron R Leiby o[]xxxx}{>>>>>>>>>>>>>>>>
Orbital **Member**	posted 07-29-99 04:02 AM I don't believe I am qualified in the least to comment about game realism, however, as an avid gamer, I do read a lot about realism in games, especially the newer ones coming up. I think a large part of making a "realistic" game (at least in my eyes, from the eyes of an avid gamer) are two things: physics and animation. At the pace gaming technology is going (or even currently), I think it is clear that textures, skins, and any other visual part of bringing objects to life won't be the limiting factor. Game realism to me lies in how convincing the models move and act. Quake 2, for example, is about as unrealistic as you can come. Why, though? The things that stick out are the model animations and the physics. When you're strafing and shooting, your feet don't move...that is blatant. Also, strafe jumping can get pretty ridiculous...another unrealistic feature that stands out. However, it looks as though there will be technology in the near future that allows MORE realistic animations, and hardware will probably come out that allows more realistic physics. (when I say physics, I don't necessarily mean only how objects behave in the world, I also mean lighting, sound, geometrical effects, like curved surfaces) Team Fortress 2 claims they are using "MRM" and skeletal technology (or something to that effect, I don't know the technical terms) for more realistic animations of the models. Apparently, instead of having predefined models, every model will be "scalable" meaning that at farther away, there will be less polygons, and up close, there will be more detail. I'm not quite sure, but it appears to me that things farther away will become more "blurred" (like in real life) and very detailed up close. The other thing, the skeletal animations also look intriguing. Instead of having predefined animations too, they can blend many different animations together, so if you're strafing, shooting, and getting hit by a bullet all simultaneously, it will blend an appropriate amount of each animation, instead of only doing one of the three. That to me sounds very, VERY realistic. Obviously I haven't seen it yet though...so I can't say for sure. ☺ 3d sound also begins to become a factor as to how convincing something is. While TF2 may or may not support the Aureal 2.0 sound that some soundcards use, I have read that half-life and heretic 2 sound phenomenally better with hardware acceleration. When you're in a hallway, it SOUNDS like you're in a hallway, when you're outside, it sounds different. When you turn a corner, a blaring siren becomes that much quieter. Or if the source of the sound is behind something, the sound is either muffled or cut off completely... Anyway, this is just my views of what realistic gaming CAN be, I am not claiming to know anything, this is all just from stuff I read, and what I get out of reading it. ☺ Orbital
Overlord **Member**	posted 07-29-99 11:27 AM First of all I want to say how glad I am with the amount of responses to my original post. Quite a few interesting things have been said and a few important questions have been raised. This post is both an assimilation of all the previous post and an expression of a few of my own thoughts. As such it is once again rather long, but I hope it is enjoyable nonetheless. Some of the points and questions can be attributed to the question of what the definition of "realism" really is. Is it the way in which we perceive (part of) that which we call "the real world"? Or is it the way in which we can be made to believe that something might be (part of) a reality? Or is it something else altogether? In the first case a higher sense of realism would result in (something very similar to) an architectural walkthrough. An example of this would be the project aiming to create a realistic representation of the Notre Dame with the Unreal engine (http://www.digitalo.com/deleon/vrnd/index.html). In the second case an improvement in realism would result in a more convincing setting, be it fantasy, sci-fi or even real world based. In my opinion this second case is how it pertains to computer games. Not only must it make everything part of a whole (one living world), but also must it make that whole breathe with the possibility of things being as they are (does that still make sense to you). In other words, not only must the blue-leafed tree made solid in the world (by utilising techniques like lighting, texturing and shading), but it must also be made conceivable that a tree can have blue leaves (by the surrounding foliage or by having a constant palette in the area). Second is the question of what we are trying to achieve. Of course (computer) games should aim at generating an

	incredible, entertaining experience. Aside from the aspects computer games have in common with movies, one of the ways to achieve this is by having a great (dynamic) gameplay. A minor part of this can be influenced by realism. For example a thick fog to limit visibility so you never really know where an attack is coming from (of course this does not have to be realistic). Another way to achieve this is to create an immersive environment, because immersion gives the feeling of really being in the world, which in turn makes the user more involved. Among other things, immersion can be achieved by freedom (large world, high level of interactivity, minimising direct linearity, etc.), consistency (similar palettes, persistence of decals, etc.), atmosphere (weather, day/night, again decals, etc.) and sound (including music). In this graphical realism influences both consistency and atmosphere. Consistency can be with the real world (things working like you are used to) or internally (persistence of decals and things working the same way throughout the world). Then there is the question of what can be achieved with graphical realism. It seems as if it lessens the amount the observer has to work to believe it. In other words it increases the observer's ability to suspend his disbelieve. The gaps that still exist between the game and the real world seem to be less important because of this. And, as seen above, realism also effects consistency and atmosphere. And atmosphere influences emotions (darkness to instil fear and reddish tints to convey a sense of warmth) and consistency conveys a sense of the world being one whole (if two similar objects are part of the same world, then they must act the same way). Of course these two (increasing consistency and atmosphere and increasing suspension of disbelieve) are closely related, since consistency and atmosphere increase the believability of the world. Finally we want to know how to achieve graphical realism. It is probably best to look at each of its facets separately. Physics seems to be important, as it increases consistency. While this is usually consistency with the real world (and thus believability), I believe that deviating physics might increase internal consistency (if done correctly). For example look at the Alien in "Alien versus Predator" or look at a movie like "the Matrix". They both have little to do with "real" physics. Another important facet seems to be lighting and shadows. They can be used both to increase consistency (for instance shadows connect objects to the world) and to increase atmosphere (there might be something lurking in the shadows). I believe that because they affect both aspects it is that they are so important to realism. But they are also very static (most of the time) which is why in a very animated medium animation is also very important. Of course you can also animate light and shadows. We are just beginning to discover the potential of curved surfaces, but since the real world contains so much of them it seems as if they at least increase consistency with the real world. On the subject of lens flares, your eyes are sort of cameras (with lenses) and if I look at a street light at night, I see a flare. Of course this may be because I am wearing glasses, but still. As with anything, overusing them can have an adverse effect and they should be used sparingly and with care. In exploring these things I want to concentrate on the graphical side of realism. I understand that things like sound and gameplay are also important for realism. But as my project concentrates on graphical realism, including sound and gameplay would be far too much, I would like to keep focussed on graphics. I also understand that realism is not the be-all-end-all of games. In fact it seems to be more a by-product. But it is very important for immersion (where this reality is not necessarily the real world). And immersion in turn is an important part of the game experience. And I am still interested in hearing why certain things are (not) done in respect to realism and what people like to see/do/make if computers where 1000 times faster. ---------- Game development is the Renaissance art of the information age. - Ben Sawyer -= Overlord =-
Sam Brown **Administrator**	posted 07-29-99 01:06 PM There were a couple things I wanted to comment on. First, my original (and admittedly bizarre message) was supposed to advocate immersion. I didn't want to jump out and say that, but it was my thesis. Immersion has the effect of allowing your brain, almost effortlessly, to fill in the gaps that the computer doesn't or can't represent. If you are fully immersed in a virtual environment, even a flat shaded untextured wall can appear totally real. However, realism (graphical, aural, spatial, etc.) doesn't individually or even collectively create immersion, and I think it's that immersion that defines how real something seems to the observer. Can you make something appear real with today's technology? Probably, can you make it FEEL real? That's tougher. Technology doesn't define how real it is, the observer does. This is why it's so important to understand your audience, it's not the game designer that determines how real something feels, it's the person playing the game. When you asked in your original message what aspects of the game make it more realistic you listed off a series of technologies/applications that you think affect realism. You also ask for ways in which the aspects are realised, basically, how they are applied or implemented to produce a sense of realism in the graphic or visual domain. And you know I just can't answer that. (BTW, I'm not trying to suggest I'm a creator or artist here...just using a perspective.) Can you ask an artist to explain his inspiration or a poet to explain the emotion that produced the piece? Designing an immersive gaming experience is about feeling, it isn't a mechanical process of adding lights to create the right shading, using realistic seeming architecture or adding lens flares where they are appropriate, it's knowing how to combine all those elements on the artist's palette and then applying the paint in such a way that the observer will feel immersed. I don't know, maybe the level designers can explain that to you, but in my experience in talking to them, they have this innate way of just knowing how something should look and feel. Some people can do that very well and others just don't get it. I am one of those who doesn't have a feel for creating an environment, but I do know when something FEELS real, and just like any other observer, that should never be discounted as unimportant. As for the question about super fast computers and what technologies would be available to us, I would have to say visual advances are even now significantly less important than the interface. If 100 years from now we were still using an engine on the same level as Wheel of Time what would be making it feel 1000 times more realistic? The interface. Be it a virtual reality headset, little goggles, a laser tracing an image directly on to the retina or a direct neural link, any of those would provide the ultimate link to the system and thereby produce and convey an incredible sense of realism the likes of which we can't even conceive of today. Of course, I believe graphic technologies will continue to improve along the exponential curve and that's great. Larger more expansive environments, more realistic lighting (real time raytracing is one possibility), true curved surfaces, a break from standard polygons, transparency, all of that will be great, but I don't believe advances in graphics is what will drive how real something feels. If computers were 1000 times faster I think that increased processing power would be devoted to improving the man-machine interface. THAT is what will improve immersion, allowing your brain to fill in even fewer gaps in the visual elements. ----------

	--=[Sam Brown]=-- Legend Entertainment sbrown@legendent.com [This message has been edited by Sam Brown (edited 07-29-99).] [This message has been edited by Sam Brown (edited 07-29-99).]
BenPatient **Member**	posted 07-29-99 06:45 PM a question for Legend: Will "rain" look like the rain in Diakatana?? I appreciate the effect in that game, but it wasn't very realistic looking, and I got tired of it very quickly. I hope you guys have made a better rain engine... Ben
Haral **Junior Member**	posted 08-02-99 06:48 AM I agree with Sam completely btw. I actually just read the first post and got thinking to it on my own, then read through the rest and Sam expressed exactly what I was feeling. Take when you read a book. Sometimes I've heard people say they didn't like a book because they couldn't "get into" it... or be immersed in the story. However when you are what you're reading becomes more than just words on a page, but rather images in your mind. The real world is in fact pushed back and you become involved in the world in which you are reading about. The same holds true for other forms of media. Honestly, I don't really believe that graphics has much to do with the realism of the experience, beyond something that might be annoying. We've seen screenshots and small samples of play for the game, but those don't show what realism will be in the game. They may look beautiful (and I certainly think they do ☺) but that all fades behind what you are doing in gameplay. So really I'd say that the relation in graphics and realism would be how smoothly it runs.
Kaliv **Member**	posted 08-02-99 10:58 AM On the subject of realism, I was watching the movie Ransom (the DVD version mind you for all you people who want WOT to come out on DVD) yesterday and you should take a look at when the people get shot. Looks kind of fake. Why?? The blood spewing out of the body looked wrong. Kind looked like someone throwing a rock in a pond. Then again, I never experienced someone getting shot first hand...so I could be wrong. Actually, this has nothing to do with the thread really. Oh well. ☺
Mahram **Member**	posted 08-02-99 04:56 PM Good topic, Overlord! My 2.153 cents: It seems to me that realism within a game, just like Glen mentioned, doesn't seem to be the goal of games. Creating a believable environment is. If the continuing trend is to try and create worlds and visualisations in real time, as close to real-life situations as possible, and merge the two to create an interactive world that people can move through, then it will be achievable very soon. If you look at how games are "real-time" rendered, and compare it with a single frame pre-render or a pre-rendered animation, you will notice the key difference being one is playable, and one will not be playable in real-time until sometime in the future. I've seen a rendered picture of the inside of a building that was absolutely real-nothing but someone telling me it was a render could convince me otherwise. I'm sure that the same thing could be done with a real-time rendering engine, but so-and-so Quake 3 player over here just reduced the display to 640x480, and reduced the texture quality so that he could get 100 fps when playing. I don't think that he is much interested in realism. I think that in the end, that is the key difference. His computer possibly can't handle real-time rendering of pre-rendered quality images. All of this is going on the assumption that this is the final goal of current trends. Real-time rendering and interaction with pre-rendered, truly life-like quality images. Since this seems to be fast approaching, this isn't really the focus. The peripherals, on the other hand, are what make up the bulk of the game. I was salivating over Unreal when I first saw it, but I got used to the image quality pretty fast, and the gameplay aspects then took over and are what kept me playing. I think that if your peripheral qualities are top-notch, and you have visual bang, you are doing extremely well. ---------- "Here's a major point of tension in most writers' lives: How can we rub enough with the world to nourish our writing, while keeping the world enough at bay to safeguard our creative energies?" --Francine du Plessix Gray amradin@home.com

Appendix 2 – Game Genres

The following list is meant to give a simple indication of what type of games are meant with each genre. This list is by no means exclusive and by no means is the genre division as clear as this table suggests. For instance, while *Wheel of Time* is considered a shooter, it has both Adventure and Strategy elements. Yet, when thinking of one of these games, the genre presented here is what is seen as the genre for that game.

Genre	**Adventure**	**Edutainment**	**Fighting**	**God**	**Platform**	**Puzzle/Card**
Game	King's Quest Space Quest Police Quest Quest for Glory Leisure Suit Larry Maniac mansion Zack McKracken Monkey Island Indiana Jones Sam & Max Grim Fandango	Dr. Brain Redcat Where in the World is Carmen Sandiego (Sim City) (Tetris)	Karataka International Karate Mortal Kombat Street Fighter Virtua Fighter Tekken	SimCity SimWorld SimAnt SimTower Sims Populous	Donkey Kong Super Mario Bros Sonic Prince of Persia Mario 64 Tomb Raider Abe's Odyssey	Hearts Patience Hoyle
Genre	**Role-playing**	**Shooter**	**Simulation**	**Sport**	**Strategy**	
Game	Ultima Bard's Tale Wizardry Might and magic AD&D Final Fantasy Fallout Baldur's Gate Ultima Online Diablo	Space Invaders Asteroids R-Type Doom Quake Unreal Half-life Amsterdoom Wheel of Time	Flight Simulator Falcon Flight Unlimited Need for Speed Test Drive Carmageddon Grand prix A2 Racer X-Wing Wing Commander Decent Freespace	EA Sports FIFA Soccer	Dune 2 Warcraft Starcraft Command & Conquer Myth	

Bibliography

[1] 3dfx Interactive, Inc. "3dfx Announces Breakthrough 3D Technology" *Press Release*, August 1999. October 1999, <http://www.europe.3dfx.com/view.asp?IOID=288>

[2] Advanced Rendering Technology Ltd. "AR250 Ray Tracing Chip" *Advanced Rendering Technology*, Februari 1999. November 1999, <http://www.art-render.com/en/>

[3] Argo Enterprise & Associates. "Dictionary of Art Terms." *Aliceville's Art Museum*, January 1997. September 1999, <http://www.aliceville.com/artmuseu.htm>

[4] Baraff, David, and Andrew Witkin. "Large Steps in Cloth Simulation" *Proceedings of the 25th Annual Conference on Computer Graphics*, pp. 43-54, Orlando, USA, July 1998. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/280814/>

[5] Bastos, Rui, Kenneth Hoff, William Wynn, and Anselmo Lastra. "Increased Photorealism for Interactive Architectural Walkthroughs" *Proceedings of the 1999 Symposium on Interactive 3D Graphics*, Atlanta, USA, pp. 183-235, April 1999. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/300523/>

[6] Bastos, Rui, Michael Goslin, and Hansong Zhang. "Efficient Radiosity Rendering using Textures and Bicubic Reconstruction" *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, Providence, USA, April 1997. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/253284/>

[7] Belton, Robert J. "Words of Art", 1996. September 1999, <http://www.arts.ouc.bc.ca/fina/glossary/gloshome.html>

[8] Berg, M. de, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry – Algorithms and Applications*, Springer-Verlag, Berlin, 1997.

[9] Bickell, Dan. "Quake Graphics Editing FAQ" April 1998. October 1999, <http://www.planetquake.com/skins/files/qskinfaq.zip>

[10] Cambridge University Press. "Cambridge Dictionaries online." *Elt*, September 1999. September 1999, <http://www.cup.cam.ac.uk/elt/dictionary/>

[11] Christian Technologies, Inc. "Webster's 1828 Dictionary", 1998. September 1999, <http://www.christiantech.com/>

[12] Connery, Richard. “The Unofficial Quake II FAQ” *Blues News*, April 1998. October 1999, <http://www.bluesnews.com/faqs/q2faq.html>
[13] Couto, Douglas De, and Joshua Napoli. “Binary Space Partitions, or, Doom in 20 minutes” *6.838 Lecture 13 – Proximity and Ordering I: Indexing Objects*, March 1998. October 1999, <http://web.mit.edu/napoli/www/6.838/main.html>
[14] Daikatana Cabana staff. “Daikatana Cabana FAQ” *Stormtroopers*, March 1998. October 1999, <http://www.stormtroopers.com/daikatana/faq.shtml>
[15] Delahunt, Michael R. “Realism or the Realist school and realism.” *ArtLex*, September 1999. September 1999, <http://www.artlex.com/>
[16] Diefenbach, Paul J. “Pipeline Rendering: Interaction and Realism Through Hardware-based Multi-pass Rendering”, 1996. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[17] Diefenbach, Paul J., and Normal I. Badler. “Multi-Pass Pipeline Rendering: Realism For Dynamic Environments” *Proceedings of the 1997 symposium on Interactive 3D Graphics*, Providence, USA, pp. 59-70, April 1997. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/253284/>
[18] Diefenbach, Paul J., and Normal I. Badler. “Pipeline Rendering: Interactive Refractions, Reflections, and Shadows”, 1994. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[19] Eclipse Entertainment. “Genesis3D Features” *Genesis3D Open Source project*, 1998. October 1999, <http://www.genesis3d.com/>
[20] Eliasmith, Chris. “Dictionary of Philosophy of Mind”, February 1999. September 1999, <http://www.artsci.wustl.edu/~philos/MindDict/>
[21] Ernst, I., H. Russeler, H. Schulz, and O. Wittig. “Gouraud Bump Mapping” *Proceedings of the 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Lisbon, Portugal, pp. 47-53. August 1998. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/285305/>
[22] Fisher, Scot S., and Glen Fraser. “Real-time Interactive Graphics” *Computer Graphics*, pp. 15-19, May 1998.
[23] Foley, J.D., A. van Dam, S.K. Feiner, J.H. Hughes, and R.L. Philips, *Introduction to Computer Graphics*, Addison-Wesley, Reading, Massachusetts, 1994.
[24] Gershbein, Reid, Peter Schröder, and Pat Hanrahan. “Textures and Radiosity: Controlling Emission and Reflection with Texture Maps”, March 1993.
[25] GT Interactive. “Graphical realism.” *Wheel of Time game forum*, August 1999. September 1999, <http://forums.gtgames.com/Forum6/HTML/000548.html> (See Appendix 1 on page 40)
[26] Gumhold, Stefan, and Tobias Hüttner. “Multiresolution Rendering With Displacement Mapping” *Proceedings of the 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Los Angeles, USA, pp. 55-66, August 1999. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/311534/>
[27] Haeberli, Paul. “Synthetic Lighting for Photography” *Grafica Obscura*, Januari 1992. October 1999, <http://www.sgi.com/grafica/synth/index.html>
[28] Hague, James. “From the Guest Editor” *Computer Graphics*, pp. 43, May 1998.
[29] Hardt, Stephen, and Seth Teller. “High-Fidelity Radiosity Rendering at Interactive Rates”, 1996. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[30] Harvey, Ian. “This is serious play.” *CNews*, April 1998. October 1999, <http://www.canoe.com/Tech1998/connect_0408.html>
[31] Heckbert, Paul S., and Michael Garland. “Multiresolution Modeling for Fast Rendering” *Proceedings of Graphics Interface ‘94*, pp. 43-50, May 1994. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[32] Heidrich, Wolfgang, and Hans-Peter Siedel. “Ray-tracing Procedural Displacement Shaders”, 1998. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[33] Heindrich, Wolfgang, and Hans-Peter Siedel. “View-independent Environment Maps” *Proceedings of the 1998 Eurgraphics/SIGGRAPH Workshop on Graphics Hardware*, Lisbon, Portugal, pp. 39-45, August 1998. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/285305/>
[34] Intel. “Intel® 3D Software Technologies - Multi-resolution Mesh” *Intel Architecture Labs*, 1999. October 1999, <http://developer.intel.com/ial/3Dsoftware/mrm.htm>
[35] Invisible Angel. “A Brief History of Graphics II” *PC Paradox*, 1999. September 1999, <http://www.pcparadox.com/Editorials/History2/1997.shtml>
[36] Invisible Angel. “A Brief History of Graphics” *PC Paradox*, 1999. September 1999, <http://www.pcparadox.com/Editorials/History/1994.shtml>

[37] Jean, Scott St. “Vampire – The Masquerade : Redemption Online Introduction” *Nihilistic Software*, September 1999. November 1999, <http://www.nihilistic.com/introduction.html>
[38] Jouppi, Norman P., and Chun-Fa Chang. “Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency” *Proceeding 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pp. 85-93,Los Angeles, USA, August 1999. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/311534/>
[39] Jouppi, Norman P., and Chun-Fa Chang. “Z^3: An Economical Hardware Technique for High-Quality Antialiasing and Transparency” *Proceedings 1999 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pp. 85-93, Los Angeles, USA, August 1999. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/311534/>
[40] Keller, Alexander. “Instant Radiosity” *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques*, Los Angeles, USA, August 1997. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/258734/>
[41] Kemerling, Garth. “A Dictionary of Philosophical Terms and Names”, 1999. September 1999, <http://people.delphi.com/gkemerling/dy/index.htm>
[42] Kiasma – The Museum of Contemporary Art. “Is Art a Picture of the World?” *Guide to the museum of contemporary art*. September 1999, <http://www.fng.fi/fng/html4/en/kiasma/guide/frame.htm>
[43] Kobbelt, Lief, Swen Campagna, Jens Vorsatz, and Hans-Peter Siedel. “Interactive Multi-Resolution Modeling on Arbitrary Meshes” *SIGGRAPH ’98: Proceedings of the 25th annual conference on Computer Graphics*, pp. 105-114, July 1998. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/280814/>
[44] Kruger, Gabe. “Curved Surfaces Using Bézier Patches” *Gamasutra, Vol. 3: Issue 23*, June 1999. October 1999, <http://www.gamasutra.com/features/19990611/bezier_01.htm>
[45] Kumar, Subodh. “Interactive Visualization of Triangular Bézier Surfaces” *ICVGIP 98, New Delhi, India*, 1998. October 1999, <http://www.cs.jhu.edu/~subodh/>
[46] Legend Entertainment Company. *The Wheel of Time*, September 1999. September 1999, <http://www.wheeloftime.com/>
[47] Lexico LCC. *Dictionary.com*, 1999. September 1999, <http://www.dictionary.com/>
[48] Loscos, Céline, and George Drettakis. “Interactive High-Quality Soft Shadows in Scenes with Moving Objects” *EUROGRAPHICS ’97*, Volume 16, Number 3, 1997. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[49] Mahony, Thomas, and James Railton. “The Unofficial Unreal FAQ v0.99.5” *Planet Unreal*, September 1998. September 1999, <http://www.planetunreal.com/faq/index.shtm>
[50] Matero, Lon. “The Official Prey FAQ”, February 1998. October 1999, <http://www.apogee1.com/faq/preyfaq12.txt>
[51] McNett, Bryan. “Multitexture and the Quake 3 graphics engine” *The Wacky World of Advanced Graphics Technology*, 1998. October 1999, <http://www.bigpanda.com/trinity/article1.html>
[52] Mitchell, Jason L., Michael Tatro, and Ian Bullard. “Multitexturing in DirectX 6” *Game Developer*, vol. 5, no. 9, pp. 33-37, September 1998. October 1999, <http://www.gamasutra.com/features/programming/19981009/multitexturing_01.htm>
[53] Möller, Thomas, Eric Haines, *Real-Time Rendering*, A K Peters, Ltd., Natick, Massachusetts, 1999. <http://www.acm.org/tog/resources/RTR/>
[54] Möller, Thomas. “Radiosity Techniques for Virtual Reality – Faster Reconstruction and Support for Levels Of Detail” *The Fourth International Conference in Central Europe on Computer Graphics and Visualization ’96*, Plzen, Czech Republic, February 1996. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[55] Morris, Daniel. “5 Hot New Gaming Technologies” *PC Gamer Online*, July 1999. October 1999, <http://www.pcgamer.com/>
[56] Mueller, Carl. “Carl’s Graphics Primer” *Carl’s Page*. October 1999, <http://epsilon.cs.unc.edu/~mueller/>
[57] Naoumov, Dan “Jago”. “The Unofficial Quake Arena FAQ” *TeleFragged*, 1998. October 1999, <http://home.telefragged.com/q3faq/>
[58] Nettle, Paul. “Radiosity in English II: Form Factor Calculation” *Grafix3D*. October 1999, <http://www.grafix3d.tzo.com/main/index.html>
[59] Nettle, Paul. “Radiosity in English” *Grafix3D*. October 1999, <http://www.grafix3d.tzo.com/main/index.html>

[60] NPD Group, Inc. “NPD Reports The U.S. Video Game Industry Hit An All-Time High In Annual Sales For 1998”, January 1999. October 1999, <http://www.npd.com/corp/press/press_990125.htm>
[61] NVIDIA Corporation. “Microsoft® DirectX® 7: What’s New for Graphics” *Technical Briefs: Looking Under the Hood*, 1999. October 1999, <http://www.nvidia.com/GeForce256.nsf/htmlmedia/techbriefs.html>
[62] NVIDIA Corporation. “Perfect Reflections and Specular Lighting Effects With Cube Environment Mapping” *Technical Briefs: Looking Under the Hood*, 1999. October 1999, <http://www.nvidia.com/GeForce256.nsf/htmlmedia/techbriefs.html>
[63] NVIDIA Corporation. “Transform and Lighting” *Technical Briefs: Looking Under the Hood*, 1999. October 1999, <http://www.nvidia.com/GeForce256.nsf/htmlmedia/techbriefs.html>
[64] Ofek, Eyal, and Ari Rappoport. “Interactive Reflecions on Curved Objects” *Proceedings of the 25th annual conference on Computer Graphics*, Orlando, USA, pp. 333-342, July 1998. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/280814/>
[65] Olsen, Mark & LaRowe, Gavin. “1913 Webster's Revised Unabridged Dictionary” *ARTFL Project*, January 1998. September 1999, <http://humanities.uchicago.edu/forms_unrest/webster.form.html>
[66] Parent, Rick. *Computer Animation: Algorithms and Techniques*, 1998. October 1999, <http://www.cis.ohio-state.edu/~parent/book/outline.html>
[67] PC Gameplay. “De GeForce is gearriveerd” *PC Gameplay*, jaargang 6, nummer 50, December 1999. November 1999, <http://www.pcgameplay.nl/>
[68] Persson, Michael “Saxs”. “Messiah Technology” *Shiny Entertainment’s Messiah*, 1998. October 1999, <http://www.messiah.com/>
[69] Polevoi, Robert. *3D Animation Workshop*, September 1999. October 1999, <http://www.webreference.com/3d/>
[70] POV-Team™. “Persistence of Vision™ Ray-Tracer - POV-Ray™ Version 3.1g - User's Documentation”, May 1999. October 1999, <http://www.povray.org/>
[71] Real 3D. “Real 3D Primer”. October 1999, <http://www.real3d.com/about/contents.html>
[72] Remedy Entertainment. “Max Payne – Frequently Asked Questions” *Max Payne*, October 1999. October 1999, <http://www.maxpayne.com/>
[73] Rouse III, Richard. “Do Computer Games Need to be 3D?” *Computer Graphics*, pp. 64-66, May 1998.
[74] Saint-André, Peter. “The Ism Book”, 1999. September 1999, <http://www.monadnock.net/ismbook/>
[75] Salvati, Jean. “3D Graphics Glossary” *Benchmarks for 3D Graphics Accelerators*, July 1997. October 1999, <http://www10.pair.com/jsalvati/ >
[76] Sawyer, Ben, *The Ultimate Game Developer’s Sourcebook*, Coriolis Group Books, Scottsdale, Arizona, 1996.
[77] Schöffel, Frank. “Radiosity and Raytracing” *Department Visualization and Virtual Reality*, December 1997. October 1999, <http://www.igd.fhg.de/www/igd-a4/research/radiosity/>
[78] Segal, Mark, Carl Korobkin, Rolf van Widenfelt, Jim Foran, and Paul Haeberli. “Fast Shadows and Lighting Effects Using Texture Mapping” *SIGGRAPH ’92 Computer Graphics*, pp. 249-252, July 1992. October 1999, <http://www.acm.org/pubs/contents/proceedings/graph/133994/>
[79] Sharp, Brian H. “Optimizing Curved Surface Geometry” *Game Developer*, pp. 40-48, July 1999. October 1999, <http://www.gdmag.com/>
[80] Shubinski, Robert G. “Glossary of Poetic Terms.” *Bob’s Byway*, August 1999. September 1999, <http://shoga.wwa.com/~rgs/glossary.html>
[81] Sieks, David. “Where the Sun Don’t Shine” *Game Developer*, pp. 61-66, April/May 1996. <http://www.gdmag.com/>
[82] Sims, Dave. “Serious Work: Programming for Games.” *IEEE Computer Graphics and Applications*, pp. 8-10, September 1994.
[83] Smith, Alvy Ray. “A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square, A Pixel Is *Not* A Little Square! (And a Voxel is *Not* a Little Cube)” *Technical Memo 6*, Microsoft Research, July 1995. November 1999, <http://www.alvyray.com/Memos/default.htm>
[84] Soler, Cyril, and François X. Sillion. “Fast Calculation of Soft Shadow Textures Using Convolution” *Proceedings of the 25th Annual Conference on Computer Graphics*, pp. 321-332, Orlando, USA, July 1998. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/280814/>

[85] Sprachwissenschaftliches Institut of Ruhr-Universität Bochum. *CCSD Online*. September 1999, <http://www.linguistics.ruhr-uni-bochum.de:8099/index.html>
[86] Terzopoulos, Demetri, Barbara Mones-Hattal, Beth Hofer, Frederic Parke, Doug Sweetland, and Keith Waters. “Facial Animation: Past, Present and Future” *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques (panel)*, pp. 434-436, Los Angeles, USA, August 1997. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/258734/>
[87] Weinhaus, Frederick M., and Venkat Devarajan. “Texture Mapping 3D Models of Real-World Scenes” *ACM Computing Surveys, Vol. 29: No. 4*, pp. 325-365, December 1999.
[88] White, Josh. *Designing 3D Graphics*, John Wiley & Sons, Inc, 1996.
[89] Winner, Stephanie, Mike Kelley, Brent Pease, Bill Rivard, and Alex Yen. “Hardware Accelerated Rendering of Antialiasing Using a Modified A-buffer Algorithm” *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques*, pp. 307-316, Los Angeles, USA, August 1999. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/258734/>
[90] Winner, Stephanie, Mike Kelley, Brent Pease, Bill Rivard, and Alex Yen. “Hardware Accelerated Rendering of Altialiasing Using a Modified A-buffer Algorithm” *Proceedings of the 24th Annual Conference on Computer Graphics & Interactive Techniques*, pp. 307-316, Los Angeles, USA, August 1997. November 1999, <http://www.acm.org/pubs/contents/proceedings/graph/258734/>
[91] Zagier, Ellen J. Scher. “A Human’s Eye View: Motion Blur and Frameless Rendering” *Crossroads – The ACM Student Magazine*, Augustus 1999. November 1999, <http://www.acm.org/crossroads/xrds3-4/ellen.html>
[92] Zhao, Jianmin, and Norman I. Badler. “Inverse Kinematics Positioning Using Nonlinear Programming for Highly Articulated Figures” *ACM Transactions on Graphics*, volume 13, number 4, October 1994. October 1999, <http://www.acm.org/pubs/contents/journals/tog/1994-13/>