A Technical Report on

3D Gaming Engines
& How to Program Them

Sameer Singh – 83 / ECE / 2000

May 2002

Contents

Preface

Generating realistic images of virtual environments is one of the most ambitious goals of computer game developers. The popularity of the game depends on, if non-technical spheres of game developing are excluded, only on the realism of the game. The level to which the game emulates reality directly affects the user stimuli towards the game, and henceforth affects the sales.

3D Engines, when used for gaming, allow users to experience something they would like to, but cannot, due to financial, and sometimes even moral issues. These may range from flying through the Earth's atmosphere at supersonic speeds, or chainsawing strange-looking creatures. Finances restrict us from pursuing the first, and the habit of seeking out unnatural creatures to chainsaw for entertainment purposes proved to be unfruitful and was mostly abandoned in the Middle Ages. Computer Games provide a cheap alternative, and thus, to satisfy the user, it is imperative to emulate reality as much as possible.

Of these games, by far the most popular have been the FPS, or First Person Shooters. In these games the user view is as from eyes of a normal person, and most movements of the player can be controlled. The popularity of these games is not surprising since they allow the player to move to wherever they want to, in the given world. Most of these games have their share of violence (the public obsession of which is quite evident).

Programming FPS is the most challenging of all types of 3D games (like car racing, flight simulators, etc.), both in terms of the quantity and quality of mathematics and physics involved. The concepts used range from simple tracing of rays to moving masses to dynamics of lights. These concepts are subdivided and distributed and calculated by different algorithms.

Brief introduction to these concepts and algorithms can be found in the following pages.

Abstract

Though a report on programming, it does not contain any code whatsoever. It just aims to provide the reader with the basics concepts involved and to introduce him to the various technologies used to develop an FPS game. These are the prerequisites of any game developer, and will never be outdated.

A game is made up of components, all of which running together or in sequence, give the desired result. A detail about these components, and their sequence of execution is contained in the introduction.

To explain the programming aspect of 3D engines, 2 engines are explained in detail. The first one, a raycasting engine, is a very specific engine. It's an FPS engine with very few features, and restricts the movement of the player. The details of raycasting engines are given in "Ray Casting".

A 3D engine can be used for many purposes. Thus, a generic engine is preferred which can handle most tasks required. In the third part, a generic engine is given in detail. This engine is tougher to program, but features can be added later easily, and gives a much more realistic output. It includes in brief, all the various concepts involved, problems faced and algorithms generally used in developing a 3D engine. It does not contain any code, just outlines the basic principles.

Where do the 3D engines go from here? What does the future hold? All these answers are given in the last part of the report, i.e. the "Outro".

Introduction: What is a 3D engine?

A 3D engine, by definition, is a collection of functions, variables and data files, which can be easily altered and used to display a three dimensional space on the 2D screen (like the monitor). But this is a very wide definition (covering programs used for CAD applications, to engines analyzing terrain), and we need not relate ourselves to all those engines, but only to ones particular to the gaming, and specifically, FPS.

A game, specifically an FPS game, can be divided into components. They are executed in a particular order, and their individual execution is quite independent of other components. They do not actively interact with each other, only pass various necessary parameters to each other.

The different components of a FPS game, and their interactions are given in Fig. 1. This is not a set rule of division of components, but is followed by most game developers. A few changes here and there are done only to suit a particular task.

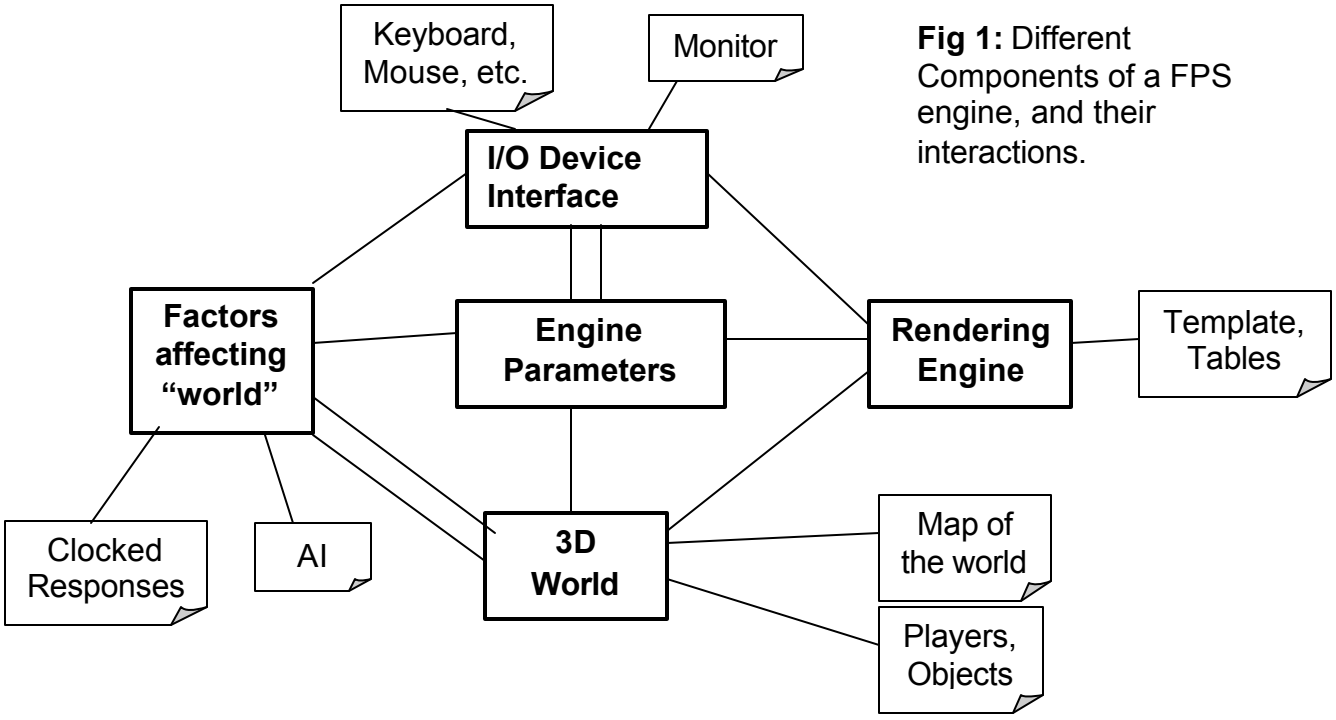


Fig 1: Different Components of a FPS engine, and their interactions.

I/O Interface: This is the software code which is used to handle the input and output devices, and convert the input to a form understandable to the engine, and convert output of the engine to the monitor mappable form. It consists of various device handlers like keyboard handlers, mouse handlers, monitor handlers, etc. which directly interact with the devices. It sends the parameters to **Factors affecting "world"** and to **Engine Parameters** while taking parameters from **Engine Parameters** and the **Rendering Engine**.

Engine Parameters: This is a group of functions and variables which store different numbers and toggles which control the engine. For example, the screen parameters (HUD, resolution, brightness, FOV, etc.), input parameters (input refresh rate, mouse acceleration, sensitivity, etc.). Thus these parameters are constantly given to various components, and can be changed only through the console.

3D World: This component basically contains data about the virtual world, and is stored in the memory in various forms. In contains the "map" of the virtual world and the static objects, like players, furniture, lights, which may change shape, or place, during the course of the game. This data can be changed only by **Factors affecting "world"** and can be accessed by the **Rendering Engine**.

Factors affecting "world": It regularly takes in data from **I/O Interface**, **3D World** and the **Engine Parameters** to calculate the various changes taking place in the virtual world, for e.g. shooting a rocket at an enemy player. Based on these inputs, necessary changes are done to the **3D World**, like create rocket object, try to make enemy dodge, show explosion, create blood on walls, and so forth. This component consists of AI and clocked responses, like explosions and other animations.

Rendering Engine: It is the busiest of all the components. It consists of calculations that convert the **3D World** data to a 2 dimensional stream, which can easily be plotted onto the output device by the **I/O Interface**. This component shall be studied in most detail.

So these are the various components of a 3D gaming engine. These are all software codes, and run together in a very complex topology, as is evident from the figure above.

Ray Casting

The raycasting engine described here is the simplest of all engines possible, and helps one to realize the ease with which a 3D engine can be developed which does not resemble the real world a lot.

The raycasting engine is used for the mostly for gaming purposes, and the one being described here can only be used effectively for FPS. In fact, the first 3D FPS for the PC, *Wolfenstein 3D*, used this engine.

In this engine, there are two basic components, the map (with all the objects) and the player. Rays are emitted by the player at some particular angle, and are traced till they intersect an object. According to the distance traveled by the ray, the object if drawn with the corresponding size. The larger the distance traveled, the smaller the object drawn. This method is shown diagrammatically in Fig. 2.

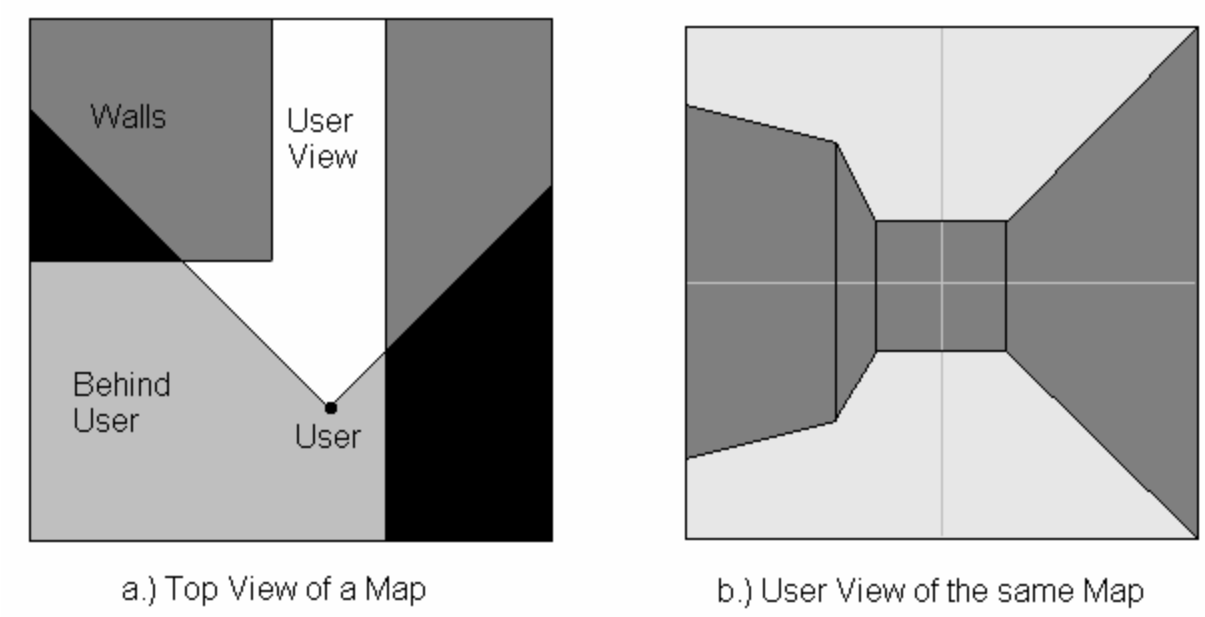


Fig. 2: Raycasting Methods

As must've been noticed, calculations involved basic trigonometry, coupled with some coordinate geometry, and the mathematics is self-derivable. This is the most primitive of all engines, but is still used for application where very fast rendering is required on very slow PCs.

Generic 3D Engine

After studying in brief the Raycasting Engine, a more generic form of a 3D engine will be explained, which is not specific only for FPS gaming. The raycasting engine was not a very real representation of the world, and was, in a way, fooling the user into believing that the world he is observing on his screen is a direct 2D representation of a 3D world. Since real physics were not used, it is very tough to add features to the engine, like lighting, unrestricted modelling of 3D objects, non-planar terrain.

In this engine, real physics are applied throughout, which eases the addition of other realistic features. The output is also highly realistic. The disadvantage, of course, is the need for faster computing, and higher memory space. The raycasting engine can be easily run on the most primitive PCs, but the following engine can only be implemented on the new generation PCs for gaming.

This engine can be easily used for CAD or terrain-generation etc. where time of rendering is not a big consideration. But in gaming, where the engine might be needed to render a world upto 70 times a second, this engine needs to be modified to make it faster. Throughout this part, after a concept is explained, it shall be analyzed to see how it can be modified for use in a FPS.

Input and Output

A 3D engine takes input from the computer, and outputs the rendered image onto the computer. Thus, though input and output devices are a very main consideration for a 3D engine, they should be undermined. This shall help the engine being developed to be used over a wide variety of platforms.

A code should be written which interacts with the I/O device, and converts data from the input device to an engine recognizable form, or vice versa for an output device. These codes are called *handlers*, and each device can have a handler of their own. Thus if devices are changed, only handlers need be changed for the engine to work as before.

For example suppose the Rendering Engine directly printed the rendered image on the monitor. This would mean if we want to change the output device (to LCD screen, printer, video recorders, etc.) we would have to make necessary changes to the Rendering Engine itself, so that it then gives an output in the form that can be displayed by the new device. This method is cumbersome, and sometimes, even impossible.

To solve this problem, the Rendering Engine stores the output image in a form known as the *bitmap*. A *bitmap* is a sequence of numbers that store the color intensities of each cell on screen. A cell of the screen is called a *pixel* (or a

picture-cell). The bitmap format is a raster format, and is used as an engine format because it is the form in which data is stored in the memory buffer of the screen. This bitmap form of the image is sent by the Rendering Engine to the output handler. If the output device is the monitor, the bitmap is directly written to the video memory, but if the output device is some other device, a handler converts the bitmap to the format to be used by the device.

The color intensity of each pixel is measured by the RGB theory, by which 3 numbers are stored. These numbers are the intensities of the colors Red, Green and Blue (hence RGB) which together are used to define any color. The values of R=0.0, G=0.0 and B=0.0 correspond to a black, while R=1.0, G=1.0 and B=1.0 correspond to white. Now if a printer is used to take the output, which uses the CMYK format, the appropriate handler converts the bitmap to a CMYK format. This thus does not interfere with the normal functioning of the Rendering Engine.

For the input devices, various variables store the different states. For example, a variable stores change in x direction (*x_dir*) and one for the weapon status (*shoot_tog*). When the user presses the forward key, the *x_dir* variable is increased by a certain value, and is then passed onto the engine, and if user presses the shoot button, *shoot_tog* is set to TRUE. If instead of keyboard, a joystick is used. Then moving the joystick forward will make the joystick handler increase the *x_dir* accordingly, and if a button is pressed, *shoot_tog* will be set. Thus to the engine it doesn't matter which input device is used since it doesn't directly interact with the device.

Handlers also keep "listening" to their respective devices, and execute themselves when an *event* occurs. Otherwise they are dormant, while the rest of the engine constantly refreshes the screen. The raycasting engine, on the other hand, renders the screen only when an *event* occurs, which maybe a key being pressed, or the world changing for some other reason.

Transformations

Real world consists of matter. This matter behaves in its own unique way, reflecting light, carrying energy of various forms, etc. These behaviors allow the humans to perceive them. This perception leads them to analyze their surroundings. But a virtual world does not exist as matter. Thus we need to define a space which can be easily managed by the computer, and when displayed to the humans, emulates reality.

For this the Euclidean space was defined, initially for two dimensions. In this space each point can be defined by three coordinates, which are the respective distance of the point from the origin when projected on the x, y and the z-axes.

All matter can be assumed to be made up of points. This is not an illogical since matter can be finely divided till a limit is reached (it may even be atomic). Once a point can be defined and stored in the memory, storing greater detail about matter is just a step further. This shall be studied in greater detail in the next unit.

These points in reality are almost never stationary, and constantly move around. The points stored in the memory also need to be changed (or *transformed*) easily. There are different types of transformations, the most basic being translation, rotation and scaling. These transformations are done using matrices, where a point is stored in a matrix ([x y z]).

Translation: Translation means moving the point to another location. It can be also think of translation as moving the coordinate system in the opposite direction. Implementing translation using matrices is somewhat complex since translation is not a linear transformation. In this transformation, the matrix used to represent a point is four-dimensional ([x y z 1]). The translated point is achieved by the following matrix multiplication.

$$\begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = \begin{bmatrix} x + t_x & y + t_y & z + t_z & 1 \end{bmatrix}$$

where t_x, t_y and t_z are the distance by which the point has to be moved in each direction. Most often translation is implemented without matrices, but by direct addition.

Scaling: Scaling is the proportional contraction or expansion of distances between points. It can also be thought of as contraction or expansion of the local space around the point. If s_x , s_y and s_z are the expansion factors in the direction of the three axes, the new point can be found out like this

$$\begin{bmatrix} x & y & z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} x \cdot s_x & y \cdot s_y & z \cdot s_z \end{bmatrix}$$

Rotation: Rotation of a point along each of the axes is a much more complex operation. Once the rotation angles, i.e. α, β and γ are decided, the rotation transformation matrix is applied.

Primitives

Once a point is defined, and different transformations are applied to it, other basic shapes also need to be defined to define a 3D world.

A point can easily be shown onto a screen. A vertex is used to define shapes containing more points. A vertex is a point that acts as a "control" point for a shape. A line, for e.g. can be defined only by 2 vertices, the starting and the ending vertex. When the line needs to be rasterized to be plotted onto a screen, other points on the line are calculated using different algorithms, like Bresenham's algorithm.

A Polygon, on the other hand, is not so simple to rasterize. A polygon can be represented either by the surrounding edges, or more easily, only by surrounding vertices. To render polygons, we must a method of scan lines. Scan lines traverse horizontally or vertically, starting from one of the boundaries, till they intersect the opposite boundary of the polygon. Enough number of scan lines is used till the whole of the polygon is rendered.

Using polygons seems like a very unrealistic practice, since flat polygons are hardly ever seen in the real world. To emulate reality better, textures, multi layering and lighting is used, which shall be studied in detail in the following pages.

Clipping

Till now, discussion has assumed the rendering of points and the primitives is flawless, that is, they shall lie on the screen. But this hardly ever the case. In 2D, points may lie outside the screen or polygons may be half on and half off the screen. In 3D, the object to be rendered may be behind the user. Thus, these have to be taken care of before actual rendering is done for correct representation of the world because we hardly ever see object behind us in real life.

The process to locate primitive's part(s) that satisfy some spatial constraints is called *clipping*.

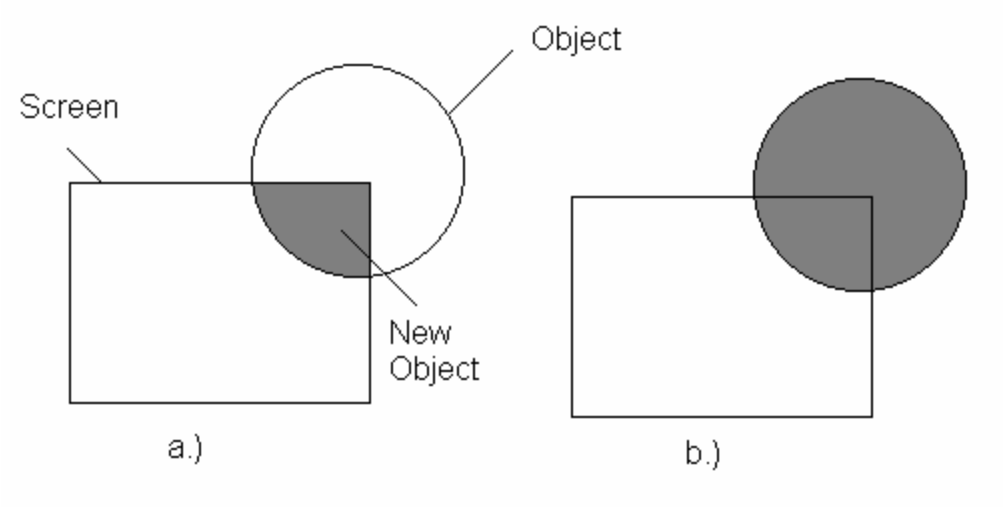


Fig. 3: 2D Clipping methods

2D or planar clipping is called *screen boundary clipping*. There are 2 basic types of planar clipping, pre- and post-rendering clipping. In the latter (Fig. 3 (b)), the object is first rendered and the part which lies on the screen is displayed. This is long and a cumbersome method. In pre-clipping (Fig. 3 (a)), the primitive is redefined keeping the screen constraints in mind. This is much faster than post-clipping, since only part of the polygon has to be rendered.

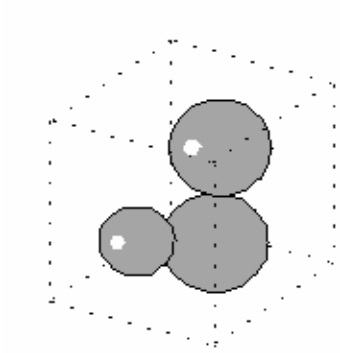


Fig. 4: Bounding Box

Though 3D objects are also based on polygons, modifying the 2D clipping techniques for volume clipping is highly calculation intensive, since in a 3D object lots of vertices and placed close together. This fact can be exploited to clip. In 3D clipping, each object is bounded by a shape of simpler geometry, and then the 2D clipping techniques are applied. Suppose an object is bounded by a box (Fig. 4), and if part of the object lies outside, we need to clip the bounding box first, after which the object can be clipped.

Rendering

This is the most important part of the engine. The methods described till now are implemented to get the final image. The rendering of a world is basically of two types, *world to screen* and *screen to world*. In the first, rays are emitted by the object and are perceived as they enter the “camera”, while in latter rays are emitted by the “camera” and are traced till they intersect any object. Raycasting engine described before is just a compact form of screen to world method of rendering.

Parameters: A rendering engine needs a few particular parameters for efficient rendering. These need to be specified before the 2 methods of rendering are discussed. These are the camera coordinates, the camera angle, and the camera focus.

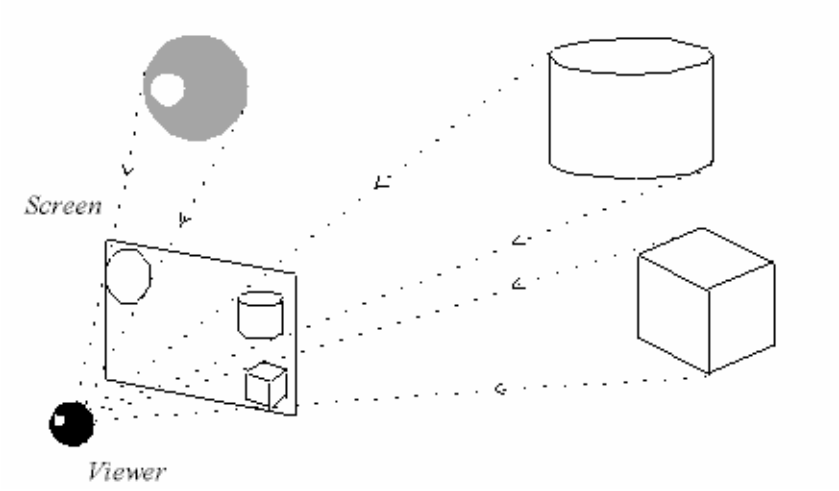


Fig. 5: World to screen method

World to Screen: This method is much slower than the second, but is preferred since it emulates reality. In the real world images are perceived by the human eyes only because light rays are reflected off objects. Thus if this method is implemented efficiently, it produces images of very realistic lighting effects. World to Screen method is implemented by regularly transforming the world space to match that of the view space. Thus it is obvious much computation power is needed. Thus method is preferred for slower games, and for applications requiring higher detail.

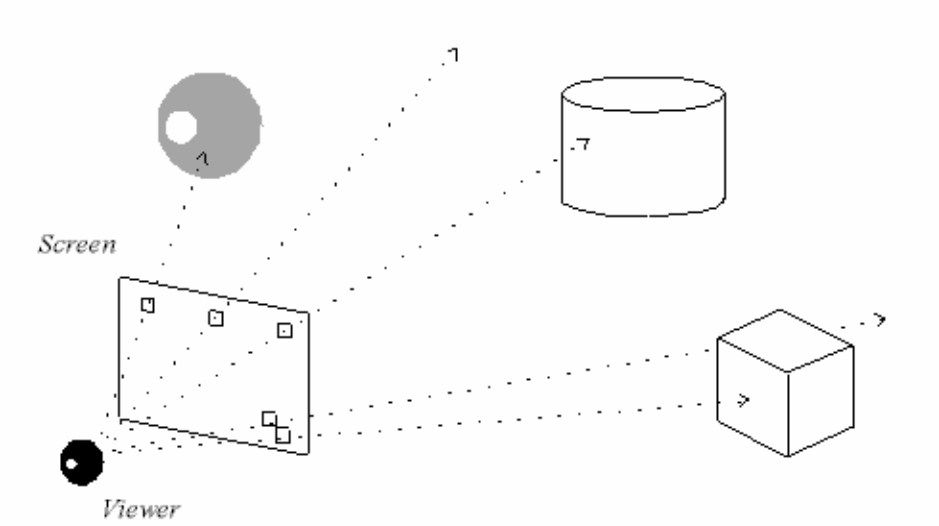


Fig. 6: Screen to World method

Screen to World: This method, though much more unrealistic, is preferred to the first for most gaming applications because of its simplicity and speed of execution. The problem of is being unnatural can be easily solved by using multilayering and light maps, which greatly enhance the quality of the output. Rays are casted from each pixel on the screen and are traced till they intersect an object, whence the color of the point of intersection is calculated. It is also preferred for outdoor worlds, where transforming huge maps is undesirable.

Modelling

Once primitives are defined, and methods of rendering a world have been discussed, what remains to be seen is the method to actually represent the world.

All models are represented by polygons. The number of polygons determine the quality of the model, that is, higher the number of polygons of an object, the more realistic the object will look. Quality of a part of an object can also be traded-off to reduce the number of polygons. For e.g., if some objects are present in the world, parts of which shall not be visible to the user, then they can be replaced by using very few number of polygons instead of wasting polygons on them.

Also, some objects may be very far of from the user all the time and the user may never be close enough to appreciate the polygons wasted on the object. Here also, fewer polygons could be used without trading off much of the quality of the overall game.

For objects that are going to be very close to the view of the user, like weapons, etc. more attention should be given. To emulate curved surfaces on these objects, splines and Bezier curves could be used, though they are highly

calculations based to be applied to every curved surface (there being so many around, especially on the human body itself).

Lighting

To make any scene appear real, it is imperative to have *some* sort of lighting system. Even the most basic form of lighting can greatly affect the visual appeal of the rendered scene. For e.g., *Wolfenstein 3D* did not have any lighting, which made it look very primitive and unrealistic, whereas *DOOM*, which came out a few months later, had just the basic lighting effects, but looked a great deal better.

Light sources can be broadly divided into three types. First are the point sources, which illuminate in all directions, like a bulb. Then there are the spotlights, the illumination of which lies in a cone with the apex at the source. Thirdly there are directional lights, which light only in one direction. This means the light rays emanating from them are parallel, for e.g. sun can be assumed to be directional for all practical purposes.

a) b) c)

Fig. 7: Lighting Effect – a) Ambience; b) Diffuse; c) Specular

The effect that these lights have on objects can then further be divided into three more categories (Fig. 7). Ambience lighting is present everywhere, and shines every surface equally. This light is not so obvious in real world in the strictest terms of its definition, but can be partly observed when objects are visible indoors during daylight. There are not many calculations involved and intensity is just added to each pixel.

Diffuse lighting, the second type of lighting, comes from point sources like the bulb. In this lighting, the change in the position of the viewer does not affect the lighting of the object; i.e. the lighting does not depend on the view angle. The intensity at each pixel of a surface depends only on the cosine of the angle between the normal to the surface and the source ($I \propto \cos\theta$).

Specular lighting emulates reality in a better way, since it keeps in mind pure reflection of the rays off objects, i.e., objects appear shiny. It depends on the view angle and if heavier on the calculations. For this reason, it is not used very often in faster games, though where speed is not a factor, using it makes a huge difference to the output. The Specular intensity at a point on the surface is proportional to the cosine of the angle between the reflected light ray and the viewer ray ($I \propto \cos\phi$).

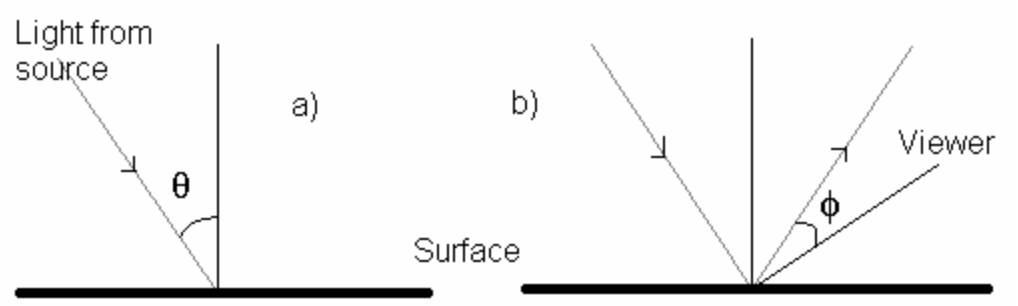
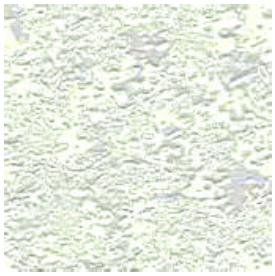


Fig. 8: Diffused Lighting (a) and Specular Lighting (b) explained using surface views

Fig. 8 (a) and (b) explain diagrammatically the angles formed in diffused and specular lighting techniques.

Texturing



Textures are basically small picture files that resemble the surface of a particular material. For e.g. the texture shown in Fig. 9 resembles that of a rough, weathered wall. Texture is applied onto polygons since flat uni-colored surfaces are not common in the real world, and even after proper lighting effects are applied, objects do not look real unless they're textured.

Fig. 9: A Texture

All polygons are now have a texture applied to them. This is known as the base texture, and this texture doesn't usually change for the course of the game. To include more dynamic surface effects, different other kinds of maps are applied *over* the base map to ensure more resemblance to reality. This is called *multilayering*. The following are some of the *maps* used.

Alpha layering: Alpha layers are monochromatic layers (i.e. they are in grayscale) which calculated the amount of transparency of a surface. These are also called masks. Till recent times constant alpha maps have been used, i.e. they are of single intensity, but grayscale masks could be used, which will vary the transparency over the surface.

Bump Maps: If surfaces are observed, small scratches and bumps can be seen, which show the roughness of the surface. This roughness though can be shown to a degree by the texture, but when they are big enough to affect the lighting on the surface, bump maps are used. These maps are also greyscale maps, with black representing a trough and white a bump. An untextured sphere will look like that in Fig. 10 if a detailed bump map is applied.

Fig. 10: Bump Map

Light maps: Calculation of diffused lighting intensities, though important, can be avoided for light sources that will not move from a position relative to the surface. For e.g., a bulb and the wall, or headlights and the bumper. Lightmaps can be colored or grayscale, and store the information about the intensity of light across a surface. Thus there is no need for the intensity to be calculated again and again. Here also, like in all multilayering techniques, we are trading memory space for the speed of the engine. An example of lightmap usage is given in Fig. 11.

(a) (b) (c)

Fig. 11: Lightmapping on a scene – (a) shows a scene with walls without any lightmap, only base textures; (b) shows the same scene with only the lightmap of a complex colored light; (c) is the output scene, after applying the lightmap to the textures.

Transparency textures: These textures are normal textures except that they have parts of themselves as transparent. These textures are then applied *over* the base textures to add details onto the base texture. For e.g., the texture of a wall is used lots of time in a game. Suppose a bloodstain is to be shown on the wall. Instead of allocating more space by creating new texture of the wall surface *with* the stain, we can just use (and later re-use) a secondary texture of a bloodstain. These maps are used for showing stains like blood, spills, etc, and for showing smaller details on the wall, like insects sitting, buttons, writing, etc.

In The End: Compiling

Once all the concepts are understood, a diagram has to be developed, which gives the tentative idea of the way the engine shall be programmed, and how the different classes and objects will be related to each other. To make this diagram, the needs and objectives of the engine have to be clearly defined, like the amount of detail that can be sacrificed for speed. The target machine should also be recognized.

After the diagram is made, based on the classes defined in it, algorithms have to be developed, implementing all the concepts. These algorithms are then coded. For coding, C++ is used most often, since it is by far the most stable and fastest language available, with compilers of almost all Operating Systems. Personally, I prefer DJGPP of all compilers, available everywhere on the net for free.

After coding, modelling of the world has to be done. This includes the 3D map, along with the different textures to be applied. Modelling of different characters is also done, and corresponding skins applied.

One of the most important parts of the engine development is testing and de-bugging. The engine should go vigorous testing and bugs should be clearly recognized and defined. This should be followed by correction of the bug (de-bugging), followed again by testing, till all the bugs are discovered.

Outro: Future

Even though 3D engine technology is only 15 years old, many developments have taken place. Each part of the 3D engine, as described in the last few pages, have been so thoroughly researched and examined that individually they are nearly perfected. If anything limits further emulation of reality, it is the restricted by the space and the speed of the currently available hardware.

The hardware dedicated for the graphics, like 3D graphics card, have also been improving very fast in the last few years, and have become an indisputable part of the configuration of a desktop now. Though a tad expensive, their capabilities justify the price. When the cards were first introduced, they had memory and an ALU specifically dedicated to calculating pixels. The modern cards, like the GeForce series, have built-in processors in them, which are faster than fastest processors available for the PC, 5 years ago.

AI, though not a part of the rendering engine, is also being studied in great detail for its application in various other fields. These concepts are also being applied to FPS gaming resulting in very human-like behaviors of bots, like in the Half-Life series.

Research is also taking place in body dynamics. The movement of humans, and other creatures, has to look real, and thus various new algorithms are being developed by the day. Most digital movies have intense mathematics involved, which can be streamlined for their application to gaming.

The 3D hardware, when combined with streamlined programming and innovative ideas borrowed from other fields will soon result in games which will erase the line between what is real, and what is not.

Summary

There are two basic factors that define the engine. First is quality, or detail. This can be measured by the resemblance of the output to real world. In quantitative terms it can be measured by the number of features available in the engine, like maximum resolution, maximum number of colors, texture size in bytes, amount of multilayering applied, type of lighting used, polygon count of bot models, etc. This factor is tougher to compare between gaming engines since they might be better in a particular field, but not so good in others.

The second is a much simpler factor, namely speed. It is measured in frames per second (also acronymed *fps*). The higher the fps, the better the game.

According to the objective of the engine, one of these factors is traded off for the other, or a balance is maintained for proper functioning. For e.g., a very fast paced game, where detail, even if present, cannot be appreciated, having higher detail is very pointless. Similarly, if a slow game is to be developed, where refresh rates of the screen are relatively low, then also it is quite pointless increasing rendering speed at the cost of quality.

Keeping the objectives in mind, proper algorithms and rendering methods should be used to program an engine that satisfies most needs.

References

Online References and Resources

Flipcode:
www.flipcode.com – A collection of tutorials and codes for game developers. The tutorials are highly informative, though thorough knowledge of underlying principles is assumed.

Game Programming with DJGPP:
www.geocities.com/SiliconValley/Park/8933 – Describes almost all the necessary details about game engines, and how DJGPP compiler can be used, along with the Allegro graphics library to develop games. Information needs updating, though

3D Coordinate Geometry:
http://www.cs.bham.ac.uk/~slb/sem307/g66.html – Text on the basics of 3D geometry, and coordinate systems. Can be utilized to further develop the concept of intersection of primitives, and transformations.

Game Developer:
www.game-developer.com – One of the most popular sites of developing games. Includes news and updates on the latest technology. Has a large collection of tutorials for understanding engine technology, and a very active mailing list.

Published Books

Tutorial: Computer Graphics: *Beatty, J.C. and Booth, K.S.* – IEEE Computer Society Press.

Computer Graphics: A Programming Approach: *Steven Harrington* – McGraw Hill

Texturing and Modelling: *Various* – Academic Press, Professional.

3dgpl's 3D Graphics Reference: *Sergei Savchenko.*