Vrije Universiteit of Amsterdam
Faculty of Sciences
Department Computer Science
Bachelor project

Amsterdam, The Netherlands June 2004

3D Gaming

S.V. Bhikharie

S.V. Bhikharie
3D Gaming
Student number: 1203940
Publication date: June 2, 2004
E-mail of author: svbhikha@cs.vu.nl
URL of author: http://www.cs.vu.nl/~svbhikha/papers

Abstract

Videogames have become a common occurrence in today's world. In this paper I have looked at what makes 3D gaming possible. To understand 3D videogames, the basics of 3D graphics are the starting point. To make a videogame, you need to understand how a videogame works. When making a videogame, a layered approach is followed. The layered structure of a videogame tells how it is built up and thus what you need to know to make one.

Table of Contents

1. Introduction

Videogames have become a common occurrence in today's world. Just like other software, it has evolved over the years. For videogames the biggest and most noticeable change has been made graphics wise. The first videogames were in 2D i.e. in a two-dimensional plane. Today's videogames mostly are in 3D. The step from 2D to 3D graphics was made possible by technological progress in the computer industry. In this paper I want to look at what makes 3D gaming possible. Therefore I have the following questions:

- What does it take to make a 3D videogame?
  - What are basic concepts for 3D graphics?
  - What are basic concepts for a videogame?
  - How do you put together the knowledge gained from the previous questions to make a 3D videogame?

I will devote a chapter per sub question. In the conclusion I will summarize the discussed material and give answers to the above questions.

2. Basic concepts of 3D graphics

According to (Watt et al, 2001) the role of three-dimensional computer graphics in the context of computer games is to supply tools that enable the building of worlds, populating them with characters and objects, having these games objects interact with each other and 'reducing' the action to a two-dimensional screen projection in real time. None of this can be accomplished without knowledge of the basic mathematics of three-dimensional space. In this chapter I will talk about these basic mathematics, but I will not go into gory mathematical details. Instead, I will give a basic overview that serves as an introduction towards understanding 3D graphics in the context of 3D videogames.

2.1. 3D coordinate system

It all starts with a three-dimensional coordinate system. This system has three axes: x, y and z. There are two common ways to arrange the (positive) axes: the right-handed and the left-handed system (figure 1 and 2).

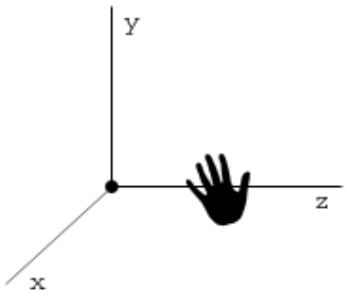

Figure 1 Left-handed coordinate system

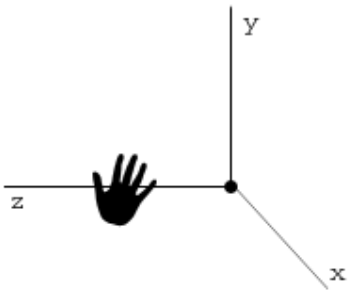

Figure 2 Right-handed coordinate system

To determine with which system you are dealing, visualize wrapping one of your hands around the z-axis. Then look at the direction your thumb is pointing. If it points in the same direction, then the hand you used is also the name for the system. If not, the name for the system is the name of the other hand.
Although right-handed systems are the standard mathematical convention, both systems are used. The difference between the two systems is the position the (positive) z-axis points to.

2.2. Objects

With the coordinate system in place, it is time to move on to the next topic: objects. At their lowest level, three-dimensional objects are represented as a set of points in the coordinate system. For simple objects you could define points in the three-dimensional space and draw lines between them to form the object. For complex objects this would be a tedious job; therefore you need a different way to represent an object in 3D. In (Watt et al, 2001) a number of mainstream computer graphics models are mentioned:

1. Polygonal model.
   Objects are approximated by a net or mesh of planar polygonal facets (polygons). In (Angel, 2003) a polygon is described as an object that has a border that can be described by a line loop, but has an interior. That is also the reason that it is sometimes referred to as fill area. With polygons you can represent objects to an accuracy of choice. More accuracy comes at the cost of using more polygons.

2. Bi-cubic parametric patches.
   These are ‘curved quadrilaterals’. Generally it can be said that the representation is similar to the polygon mesh except that the individual polygons are now curved surfaces. Each patch is specified by a mathematical formula that gives the position of the patch in three-dimensional space and its shape. This formula enables us to generate any or every point on the surface of the patch. The shape or curvature of the patch can be changed by editing the mathematical specification. This results in powerful interactive possibilities. The problems are, however, significant. When the shape of individual patches in a net of patches is changed, there are problems in maintaining ‘smoothness’ between the patch and its neighbours. Bi-cubic parametric patches can either be an exact or an approximate representation. They can only be an exact representation of themselves, which means that any object can only be represented exactly if the shape corresponds exactly to the shape of the patch. This somewhat torturous statement is necessary because when the representation is used for real or existing objects, the shape modelled will not necessarily correspond to the surface of the object.
   A significant advantage of the representation is its economy. When compared to the polygonal model, the bi-cubic parametric patches model uses fewer elements to create the same object.

3. CSG (constructive solid geometry).
   This is an exact representation to model objects within certain rigid shape limits. It has arisen out of the realisation that a lot of manufactured objects can be represented by ‘combinations’ of elementary shapes or geometric primitives. For example, a chunk of metal with a hole in it could be specified as the result of a three-dimensional subtraction between a rectangular solid and a cylinder. Connected with this is the fact that such a representation makes for easy and intuitive shape control – it can be specified that a metal plate has to have a hole in it by defining a cylinder of appropriate radius and subtracting it from the rectangular solid, representing the plate. The CSG method is a volumetric representation: shape is represented by elementary volumes or primitives. This contrasts with the previous two methods that represent objects using surfaces.

4. Spatial subdivision techniques.
   With this technique the object space is divided into elementary cubes, known as voxels and each voxel is labelled as empty or as containing part of an object. It is the three-dimensional analogue of representing a two-dimensional object as the collection of pixels onto which the objects projects. Labelling all of three-dimensional object space in this way is clearly expensive, but it has found applications in computer graphics. This model thus represents the three-dimensional space occupied by the object; the first two mentioned methods are representations of the surface of the object.

5. Implicit representation.
   Implicit functions are occasionally mentioned in texts as an object representation form. An implicit function is, for example:
   $x^2 + y^2 + z^2 = r^2$

   which is the definition for a sphere. On their own they are of limited usefulness in computer graphics because there are a limited number of objects that can be

represented in this way. Also, it is an inconvenient form as far as rendering is concerned. However, it should be mentioned that such representations do appear quite frequently in three-dimensional computer graphics – in particular in ray tracing where spheres are used frequently – both as objects in their own right and as bounding objects for other polygon mesh representations.
Implicit representations are extended into implicit functions that can loosely be described as objects formed by mathematically defining a surface that is influenced by a collection of underlying primitives such as spheres. Implicit functions find their main use in shape-changing animation; they are of limited usefulness for representing real objects. The method does show potential for modelling organic shapes. Metaballs (blobs) is an implicit modelling technique that is used for creating the aforementioned organic shapes.

From all the object models, the polygonal model is the de facto standard. The main reason is that it has low CPU requirements; however you have the accuracy/polygon count trade-off. Other object models like bi-cubic parametric patches and spatial subdivision techniques (voxels) are getting used more frequently for 3D games. Although these models have higher CPU requirements, their advantages over polygons are enough reasons to start using them. The emergence of faster CPUs and more available memory will also stimulate the usage of other available models.

2.3. Transformations

In (Watt et al, 2001) transformations are described as important tools in generating three-dimensional scenes. They are used to move objects around in an environment, and also to construct a two-dimensional view of the environment for a display surface. The basic idea behind a transformation is mapping a point to another point using some kind of function. The possible functions correspond to the different kinds of transformations. The transformations used for computer graphics are called affine transformations. According to (Weisstein, 2004), an affine transformation is any transformation that preserves collinearity (i.e., all points lying on a line initially still lie on a line after transformation) and ratios of distances (e.g., the midpoint of a line segment remains the midpoint after transformation). In this sense, affine indicates a special class of projective transformations that do not move any objects from the affine space to the plane at infinity or conversely. From a computer graphics perspective this means that an object can still be represented in the user defined 3D coordinate system after a transformation.
Affine transformations can be represented by a matrix and a set of affine transformations can be combined into a single overall affine transformation by combining the separate matrices. This makes it possible to apply different kinds of transformations to a single object. For more information on the mathematical background of affine transformations and matrices, see (Angel, 2003).
The most basic and commonly used affine transformations in computer graphics are translation, rotation and scaling. In (Angel, 2003) shear is also presented as a basic affine transformation, because of its importance. However, since it can be created by applying a sequence of the aforementioned three basic affine transformations, I do not see it a basic affine transformation. I will now give short descriptions of translation, rotation and scaling. These are abbreviations from (Angel, 2003).
Translation is an operation that displaces points by a fixed distance in a given direction. Since we are dealing with a 3D coordinate system, the translations take place in the three axis directions.

Rotation is in an operation that displaces a fixed point around a certain vector at a certain angle. For a given fixed point there are three degrees of freedom: the two angles necessary to specify the orientation of the vector and the angle that specifies the amount of rotation about the vector.
Scaling makes objects bigger or smaller in any (combination) of the three dimensions. Scaling transformations have a fixed point. To specify a scaling, you need to specify the fixed point, a direction in which you want to scale and a scale factor.

2.4. Summary of basic 3D graphics concepts

To understand 3D videogames, the basics of 3D graphics are the starting point. Firstly, a 3D coordinate system needs to be defined. Secondly, you need to choose how to represent you objects in your coordinate system. You can choose from different kinds of object models, each with their trade-offs. Currently the polygon (mesh) model is the de facto standard, but this is changing as faster CPU's and more memory become available. Lastly, you need a way to manipulate the objects in your coordinate system. This can be done with affine transformations. The three most basic and commonly used affine transformations for computer graphics are translation, rotation and scaling. Any other affine transformation can be created by applying a properly chosen sequence of the aforementioned basic transformations.

3. Basic concepts of videogames

To make a videogame, you first need to understand how a videogame works. The first thing you need to realise is that a videogame is just a program that can be compiled and executed. Just like any other program it has a main function that is the first function to be called upon when it is executed. The structure of the main function can be represented as a flowchart diagram, as depicted in figure 3 (Kuffner, 2002).

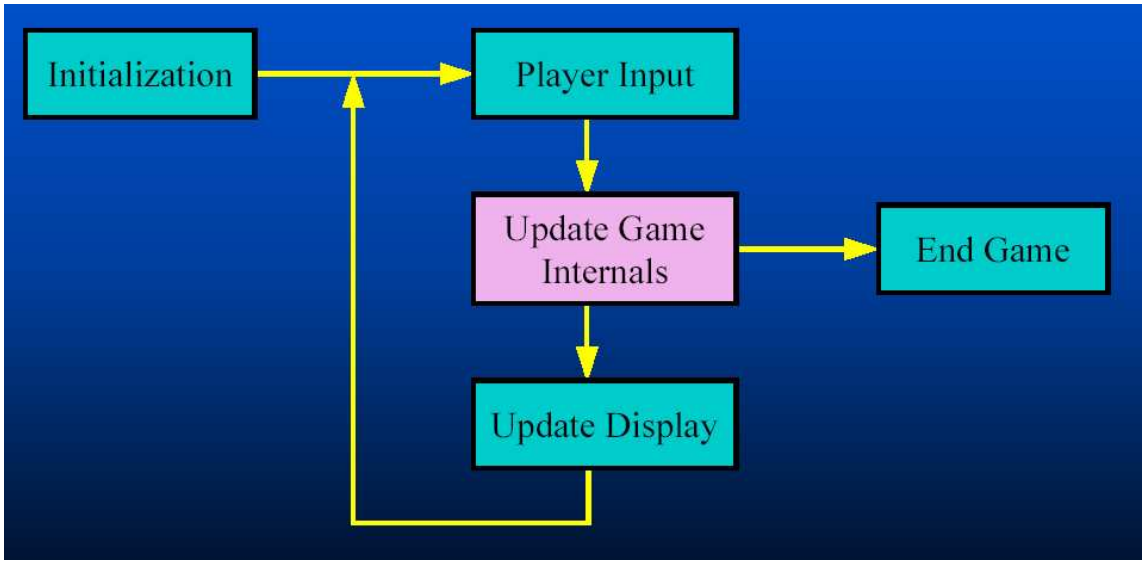

Figure 3 Flowchart diagram of the main function of a videogame

The diagram above consists of five states. The three most important states for a videogame are connected to each other in a loop. This loop is often referred to as the "game loop". In (Howland, 2001) the game loop is described as a series of procedures for getting input and displaying output to the player and updating the game. Let us now look at each state and their meaning:

- Initialization (Starting the game)
  According to (Howland, 2001) the initialization is as follows: "The beginning of every game usually consists of an animation sequence to show off some aspect of the game's story or background and an option menu for starting the game or changing various parameters that affect the game in some way. Those parameters often include sound volume, graphic options, multiplayer options, difficulty and starting level."
  I agree that this is the way a videogame is started when you play it. But when think about it, these things are actually done in the so-called game loop, because the player inputs actions and the videogame responds by performing the appropriate action. The changes have to be updated internally and have to be outputted to the player. What I think is meant with initialization is that the necessary parameters are set, functions are called et cetera that are necessary so that the videogame can move from this state to the game loop. Examples of what could be done in the initialization state are checking and requesting memory, setting up a stack, load display drivers.
- Player input
  (Howland, 2001) says the following: "The input gathering routines will take the player's input from whatever device they are using and store it in a way that the game

can process it to make changes to the game internals." Note that the stored input information is used in the next state (updating game internals).

- Updating game internals
  Here is what (Howland, 2001) has to say: "The game updating routines are the real guts of the game. Everything from moving the player's character using their input, to the actions of the enemies and determining whether the player has won or lost the game is determined here." Also, preparations are made for displaying the graphics.
- Displaying the screen
  According to (Howland, 2001) the screen can be displayed in two ways: "You can either draw everything to the screen at one time or what is more commonly done, set everything up to be drawn and then draw the screen afterwards. Drawing the screen can take longer than most processes in a game so you want to do it all at one time. Determining what is necessary to be drawn can take a relatively long time with all the checks your game could have, so it's best to do this before you actually try to commit your graphics to the screen or video hardware."
- Ending the game
  (Howland, 2001) describes that normally when a game ends, an ending sequence is displayed, or at least something that says goodbye to the player. This description is in the same spirit as his description about starting a game: it describes this state from a player's point of view. Again, I think that something else is meant, namely calling functions and setting values et cetera, so that the videogame can be terminated in a normal way.

Now that I have given an overview of the how a game basically works, it is time to look at some interesting aspects from the game loop that are essential for a modern videogame. I will focus on the updating of the game internals and more specifically on gameplay and artificial intelligence of a videogame. Before I start with discussing these subjects, I would first like to take a look at the scenario of a videogame.

3.1. Scenario

According to Webster's dictionary a scenario is a preliminary sketch of the plot, or main incidents. In the context of videogames the word scenario has the same meaning, but there is one more thing that should be taken into consideration: a videogame is interactive.
When making the scenario for a game, it is quite usual nowadays to have some kind of plot that connects the interactive elements of the videogame. Even when there is no (real) story, the scenario should describe what is possible in a videogame. This leads to the following view of what the scenario of a videogame looks like:

- The scenario at least describes the interactive elements of the videogames. This should be written down as much as possible in a non-technical way, because you are not programming (yet). These interactive elements determine the gameplay.
- The scenario can have a plot that connects the interactive elements to each other. This part of the scenario can be seen in the same way as a scenario from a movie or a play. Nowadays most videogames have a plot. For some genres of games this can lead to quite ridiculous plots, for instance puzzle games. Other genres are very dependant of the plot and cannot exist without it. In this case the plot is equally and sometimes more important than the interaction. An example is the genre of role-playing games. In this case you will need a system to present the story to the players. There are various ways for doing this, as described in (Simpson, 2002). You could for instance use cut scenes or scripting.

With the scenario (partially) developed, the gameplay can be worked out further.
You can do this by writing down more about it, or [when worked out enough] you can start with programming experiments to test it out.

3.2. Gameplay

The word gameplay is one of the keywords in videogaming today. As the word says, it determines how a game is played. Gameplay is influenced by a couple of factors:

- Scenario and genre
  In the scenario at least the interactive elements of a videogames are written down, which determine the basic gameplay form. What has not been mentioned earlier is choosing the right genre(s) for a game. Each genre brings along certain gameplay elements that may or may not suit the game scenario. It is possible to combine different elements from different genres to fit the scenario. So it is the combination of scenario and genre(s) and how it is worked out that determine the basic gameplay.
- Control and response
  Very important refinements of the gameplay are how a game is controlled and how well it responds to player input.
  The game controls should be defined according to what the player expects and knows and/or should be as natural as possible. Also, for advanced players customizable controls are recommended. A well thought up control scheme can benefit the gameplay, because it lets the game be played more easily.
  Controlling a game bring up the issue of response. The player should always be able to see what the consequences of their actions are in the videogame. The response time should always be as short as possible, according to the type of gameplay. For instance, in a race game you do not want a response time of 5 seconds when racing, because in that time span, something else has already happened.
- Viewpoint (camera)
  Although the viewpoint does not affect the gameplay directly, its choice does influence the way the game is played. If the player has a wrong view of the game world, it is harder to make a decision. For 3D games the viewpoint is often referred as the camera. The idea behind this is that the camera follows the player to show the gaming world.
- Learning curve and ease of play
  The last two factors of influence for gameplay are the learning curve and ease of play. I have placed these two together because they both deal with the level of difficulty in a game.
  The learning curve is the initial difficulty encountered when first playing a game. This affects the gameplay in such a way that only if the player knows the basics about the game, he/she can effectively play it. Thus if the player do not know enough, the gameplay is reduced, since certain gameplay elements are missed.
  The ease of play is the difficulty you engage in the game after the learning curve. The player at least knows how the basic gameplay elements work. This does not mean that the player has mastered a game. In many games it is possible to set the difficulty. This usually affects how the game world responds to the player. In the next paragraph about artificial intelligence I will come back on this.

Because gameplay is the core of each game, it will determine if the player wants to play it or not. Where graphics are dependant of technology and show their age over time, well thought out gameplay can last forever.

3.3. Artificial intelligence

According to (Kelly, 2003), Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In videogames AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game (Kelly, 2003). This should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. A so-called non-playable character (NPC) is an example of this. There are different kinds of AI techniques available that can be used for videogames. Three of these [commonly used] techniques are search algorithms, neural networks and finite state machines. In the following paragraphs I will discuss their uses in videogames.

3.3.1. Search algorithms

Search algorithms can be used to solve path-finding problems. An example of a path-finding problem in a videogame is an opponent who tries to locate the player.
(Kelly, 2003) discusses search algorithms as having a problem space model, which is the environment where the search is performed. This consists of a set of states of the problem and a set of operators that can alter the state. Although brute force search algorithms like depth-first and breath-first search always find their goal, they are usually not suitable for videogames, since they can require lots of memory and are inefficient as they can explore a large number of possible states until the goal state is reached. The A* (A star) algorithm is also a path finding algorithm, but unlike the brute force algorithms it is more efficient. For more information about A*, see (Patel, 2003).
Another search algorithm is the minimax algorithm. It looks ahead to try and anticipate what the opponent will do. This information is used to choose the best move that gives the computer the advantage (Kelly, 2003).
Whichever search algorithm is used depends on the videogame. The trade-off that has to be made is speed versus accuracy. For most games, you don't really need the best path between two points. You just need something that is close. What you need may depend on what is going on in the game, or how fast the computer is. (Patel, 2003) talks about this trade-off in the context of A*, but it can be generalized to search algorithms in general.

3.3.2. Neural networks

When talking about neural networks or neural nets (NN) the first question of course is: what is a neural network? (Sarle et. al., 1997) gives us the following answer: “There is no universally accepted definition of an NN. But perhaps most people in the field would agree that an NN is a network of many simple processors (‘units’), each possibly having a small amount of local memory. The units are connected by communication channels (‘connections’), which usually carry numeric (as opposed to symbolic) data, encoded by any of various means. The units operate only on their local data and on the inputs they receive via the connections. The restriction to local operations is often relaxed during training.”
This definition brings up an interesting aspect that makes NN interesting for use in videogames: its ability to learn. (Kelly, 2003) justly remarks that this feature makes them an interesting addition to any game as they can gradually adjust themselves to provide a more challenging experience for the player.
About the training rules (Sarle et. al., 1997) has the following to say: “Most NNs have some sort of ‘training’ rule whereby the weights of connections are adjusted on the basis of data. In

other words, NNs "learn" from examples, as children learn to distinguish dogs from cats based on examples of dogs and cats. If trained carefully, NNs may exhibit some capability for generalization beyond the training data, that is, to produce approximately correct results for new cases that were not used for training."
(LaMothe, 1999) gives a couple of examples of NN videogame applications:

- Environmental scanning and classification.
  A neural net can be feed with information that could be interpreted as vision or auditory information. This information can then be used to select an output response or teach the net. These responses can be learned in real-time and updated to optimize the response.
- Memory.
  A neural net can be used by game creatures as a form of memory. The neural net can learn through experience a set of responses. Then when a new experience occurs, the net can respond with something that is the best guess at what should be done.
- Behavioral control.
  The output of a neural net can be used to control the actions of a game creature. The inputs can be various variables in the game engine. The net can then control the behavior of the creature.
- Response mapping.
  Neural nets are really good at 'association', which is the mapping of one space to another. Association comes in two flavors: autoassociation, which is the mapping of an input with itself and heterassociation, which is the mapping of an input with something else. Response mapping uses a neural net at the back end or output to create another layer of indirection in the control or behavior of an object. Basically, we might have a number of control variables, but we only have crisp responses for a number of certain combinations that we can teach the net with. However, using a neural net on the output, we can obtain other responses that are in the same ballpark as our well defined ones.

(LaMothe, 1999) remarks that the above examples may seem a little fuzzy, and they are. The point is that neural nets are tools that we can use in whatever way we like. The key is to use them in cool ways that make our AI programming simpler and make game creatures respond more intelligently.

3.3.3. Finite state machines

In (Brownlee, 2002) finite state machines are described as follows: "Finite state machines (FSM), also known as finite state automation (FSA), at their simplest, are models of the behaviors of a system or a complex object, with a limited number of defined conditions or modes, where mode transitions change with circumstance. Finite state machines consist of 4 main elements:

- States, which define behavior and may produce actions
- State transitions, which are movement from one state to another
- Rules or conditions, which must be met to allow a state transition
- Input events, which are either externally or internally generated, which may possibly trigger rules and lead to state transition

A finite state machine must have an initial state, which provides a starting point, and a current state, which remembers the product of the last state transition. Received input events act as

triggers, which cause an evaluation of some kind of the rules that govern the transitions from the current state to other states. The best way to visualize a FSM is to think of it as a flow chart or a directed graph of states, though there are more accurate abstract modeling techniques that can be used." Figure 4 shows a picture of a FSM by (Brownlee, 2002).

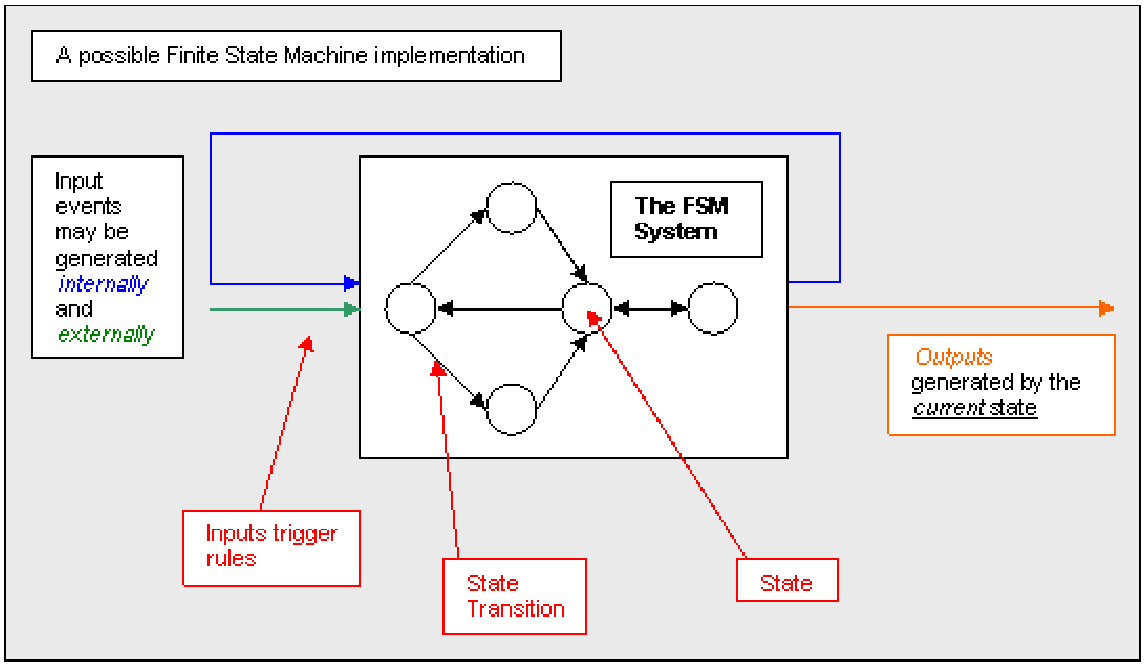

Figure 4 A possible finite state machine control system implementation

Now that we know what a FSM is, let us look at how we can use it for a videogame.
A few examples of videogames that use FSMs are the first person shooter (FPS) games Doom, Quake and Quake 2, which were all made by the same company, id Software. (Matthews, 2000) has an interesting example dealing with how the bots (NPC opponents) behave in his Quake 2 mod(ication). I have used his original FSM diagram to make figure 5.

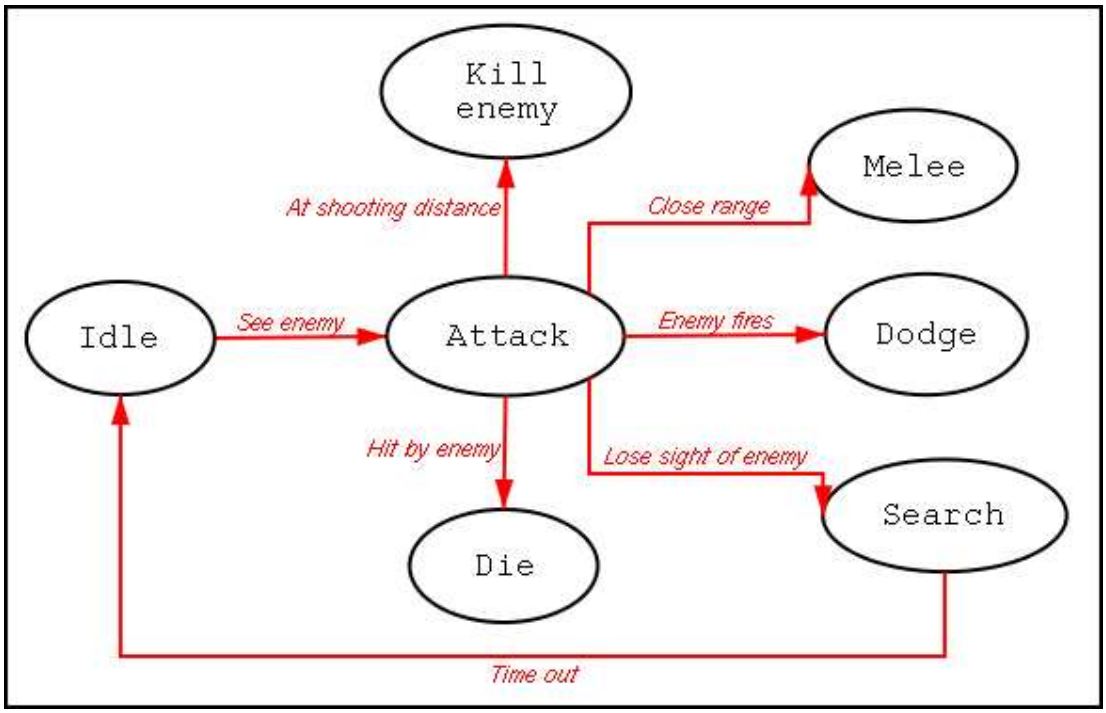

Figure 5 Finite state machine of a Quake 2 mod

Let us take a look at the behavior of the bot. Firstly the bot will be in the idle state, doing nothing. When he sees the enemy, it will go to the attack state. Depending input events the bot receives while in this state, the bot will choose the appropriate follow-up action. Although figure 5 is a simplified model, it does make clear how a FSM works.
Finite-state machines are a good way to create a quick, simple, and sufficient AI model for the games it is incorporated in (Matthews, 2000). This also has its disadvantages. (Brownlee, 2002) justly remarks that for computer games, easily predictable behavior is usually not a wanted feature, as it tends to remove the 'fun-factor' in the game. He mentions possibilities to extend the FSM to make it harder to predict (NPC) actions: "A number of extensions to finite state machines such as random selection of transitions, and fuzzy state machines shows us another common type of FSM called non-determinist where the systems actions were not as predictable, giving a better appearance of intelligence."

3.4. Summary of basic videogame concepts

To make a videogame, you first need to understand how a videogame works. The first thing you need to understand is that a videogame is just a program with a main function. The main function consists of five states. The three most important states for a videogame are connected to each other in the game loop, which is a series of procedures for getting input and displaying output to the player and updating the game. Some interesting aspects from the game loop that are essential for a modern videogame are gameplay and artificial intelligence of a videogame. However, it all starts with the scenario.
The scenario should describe what is possible in a videogame. It should at least describe the interactive elements of the videogames, which is the base for the gameplay. Furthermore, the scenario can have a plot that connects the interactive elements to each other. Nowadays most videogames have a plot. With the scenario (partially) developed, the gameplay can be worked out further.

Gameplay determines how a game is played. Gameplay is influenced by a couple of factors: scenario and genre, control and response, viewpoint (camera) and the learning curve and ease of play. Because gameplay is the core of each game, it will determine if the player wants to play it or not.
Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In videogames AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game. This should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. There are different kinds of AI techniques available that can be used for videogames. Three of these techniques are search algorithms, neural networks and finite state machines.

4. Making a 3D videogame

In this chapter the knowledge from the previous chapters is put together to prescribe how to make a videogame. The first thing we need is a game engine. (Simpson, 2002) makes a clear distinction between the game and game engine by making a comparison: “Many people confuse the engine with the entire game. That would be like confusing an automobile engine with an entire car. You can take the engine out of the car, and build another shell around it, and use it again. Games are like that too. The engine can be defined as all the non-game specific technology (for instance the renderer). The game part would be all the content (models, animations, sounds, AI, and physics), which are called 'assets', and the code required specifically to make that game work, like the AI, or how the controls work.”
The engine on its turn is made on top of a so-called application programming interface (API). An API is a set of functions you can use to work with a component, application, or operating system. For PCs, it provides a consistent front end to an inconsistent back end (Simpson, 2002). For game consoles both the front and back end are consistent. Nevertheless for both types of systems an API is necessary, because an API can ‘talk’ to the (graphical) hardware and make the necessary system calls.
The above described approach results in a layered videogame structure (figure 6). A layered approach means that a layer can use functionalities/possibilities from the underlying layer that are presented to it (the higher layer).

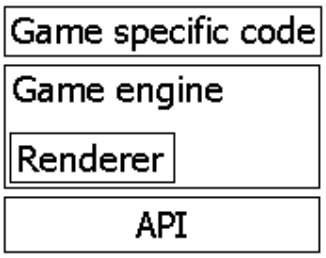

Figure 6 Layered structure of a videogame

In the following paragraphs I will discuss the three layers of figure 6 bottom-up.

4.1. Application programming interface (API)

Like stated earlier, an API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for. For game consoles each has its own specialized API(s), since the hardware does not change. For PCs the APIs need to be more general, since there is a lot of difference between the hardware of PCs. However, what you tend to see is that when a new PC game is made, it tries to take advantage of the latest hardware as possible, but at the same time it tries to support ‘older’ hardware as well. This results in a trade-off between what the minimal configuration of your PC should be versus how much of the new hardware technology can be incorporated. Currently the two major APIs used on the PC are OpenGL and DirectX. OpenGL is a pure graphics oriented API, where as DirectX is a collection of APIs for graphics, input handling, sound et cetera. The OpenGL counterpart of DirectX is Direct3D. There is an ongoing discussion on whether OpenGL or Direct3D is better. In (Roy, 2002) both are discussed through a thorough analysis without favoring one. I agree with (Roy, 2002) that you should choose for a certain API (or both!) depending on your situation and platform. In the end it does not matter which one you

choose, because once you have learnt one of them, it is should not be too hard to learn the other.

4.2. Game engine

The game engine is built on top of an API and consists of the non-game specific technology. It is very common to give a game engine the name of the first game that used it. A classic example of a game engine is the Quake engine.
The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. An engine can consist of much more than just a renderer, for instance it can have world editors, character editors, a gravity model and a collision model. There is no maximum to what an engine should contain. The examples could also be left out; it is just a matter of what you want to be in it. Keep in mind that if you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.
As mentioned earlier the renderer is the graphical component of a game engine. It is the minimum a game engine should contain. (Simpson, 2002) gives a good impression of what the renderer does: “The renderer visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed. It is generally the first thing you tend to build when constructing an engine. The renderer is where over 50% of the CPUs processing time is spent, and where game developers will often be judged the most harshly. The business of getting pixels on screen these days involves 3D accelerator cards, APIs, three-dimensional math, an understanding of how 3D hardware works, and a dash of magic dust. For consoles, the same kind of knowledge is required, but at least with consoles you are not trying to hit a moving target. A console's hardware configuration is a frozen ‘snapshot in time’, and unlike the PC, it does not change at all over the lifetime of the console.
In a general sense, the renderer's job is to create the visual flare that will make a game stand apart from the herd, and actually pulling this off requires a tremendous amount of ingenuity. 3D graphics is essentially the art of the creating the most while doing the least, since additional 3D processing is often expensive both in terms of processor cycles and memory bandwidth. It is also a matter of budgeting, figuring out where you want to spend cycles, and where you're willing to cut corners in order to achieve the best overall effect. “ For more information about how a game engine looks like, I recommend the tutorial by the earlier mentioned (Simpson, 2002).

4.3. Game specific code

The game specific code basically consists of the in the previous chapter described game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code. You can think of specific character models, sounds, special collision models et cetera. As long as it is not contained in the engine, you can add it to this layer. Other features that are usually put in the game specific code are multiplayer/networking options and HUDs/menus.
When using an existing engine for your videogame you already have a good starting point. However, it is not unthinkable that you would like to add more functionality to the engine, because it is not always possible to put that in the game specific code. Depending on how the game engine is made up, it may be possible to extend it and add the desired extra functionality.
Lastly I would like to discuss the use of game editors. It is quite impractical to create an entire world just by coding. Nowadays many games make use of specially built editors. These editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the videogame when it is released. This gives players the opportunity to build their

own worlds to play in and thus can extend the lifetime of the videogame. A step further is a so-called mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine. A very well known mod is Counterstrike, which was made based on the Half-Life engine. It even got released as a new (separate) videogame.

4.4. Summary of making a 3D videogame

When making a videogame, a layered approach is followed (figure 6). Bottom up the layers are API, game engine and game specific code.
An API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for. For game consoles each has its own specialized API(s), since the hardware does not change. For PCs the APIs need to be more general, since there is a lot of difference between the hardware of PCs. Currently the two major PC APIs used are OpenGL and DirectX.
The game engine is built on top of an API and consists of the non-game specific technology. The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. There is no maximum to what an engine should contain. If you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.
What the renderer does is visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed. It is generally the first thing you tend to build when constructing an engine.
The game specific code consists of the game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code. Other features that are usually put in the game specific code are multiplayer/networking options and HUDs/menu's.
When using an existing engine it may be possible to extend it to add extra functionality you need for your game.
Game editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the videogame when it is released. This gives players the opportunity to build their own worlds to play in. A step further is a so-called mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine.

5. Conclusion

In the introduction I asked the following questions:

- What does it take to make a 3D videogame?
  - What are basic concepts for 3D graphics?
  - What are basic concepts for a videogame?
  - How do you put together the knowledge gained from the previous questions to make a 3D videogame?

I will now give answers to each sub question.

- What are basic concepts for 3D graphics?
  Firstly, a 3D coordinate system needs to be defined. Secondly, you need to choose how to represent you objects in your coordinate system. You can choose from different kinds of object models with their trade-offs. Currently the polygon (mesh) model is the de facto standard, but this is changing as faster CPU's and more memory become available. Lastly, you need a way to manipulate the objects in your coordinate system. This can be done with affine transformations. The three most basic and commonly used affine transformations for computer graphics are translation, rotation and scaling. Any other affine transformation can be created by applying a properly chosen sequence of the aforementioned basic transformations.
- What are basic concepts for a videogame?
  The first thing you need to understand is that a videogame is just a program with a main function. The main function consists of five states. The three most important states for a videogame are connected to each other in the game loop, which is a series of procedures for getting input and displaying output to the player and updating the game. Some interesting aspects from the game loop that are essential for a modern videogame are gameplay and artificial intelligence of a videogame. However, it all starts with the scenario.
  The scenario should describe what is possible in a videogame. It should at least describe the interactive elements of the videogames, which is the base for the gameplay. Furthermore the scenario can have a plot that connects the interactive elements to each other.
  Gameplay determines how a game is played. Gameplay is influenced by a couple of factors: scenario and genre, control and response, viewpoint (camera) and the learning curve and ease of play. Because gameplay is the core of each game, it will determine if the player wants to play it or not.
  Artificial intelligence (AI) is a generic term for software that simulates an advanced level of intelligence to perform complex tasks. In videogames AI is used by game developers to add an element of realism to challenge a human opponent and enhance the quality of the game. This should provide a better and more realistic experience for the player. AI is usually applied to the elements of the game world the player cannot control or influence. There are different kinds of AI techniques available that can be used for videogames.
- How do you put together the knowledge gained from the previous questions to make a 3D videogame?
  To make a 3D videogame, a layered approach is followed (figure 6). Bottom up the layers are API, game engine and game specific code.

An API is a set of interfaces to access lower level services. The API that can be used depends on the system you want to develop a game for.
The game engine is built on top of an API and consists of the non-game specific technology. The game engine is not the same as the renderer. The renderer only makes up the graphical part of a game engine. There is no maximum to what an engine should contain. If you want your engine to be reused for other games, it (preferably) should not contain any game specific technology.
What the renderer does is visualizes the scene for the player / viewer so he or she can make appropriate decisions based upon what is displayed.
The game specific code consists of the game loop, which contains the interactive elements that make up the gameplay and the AI. Also, all the things you cannot put in the game engine are part of the game specific code.
Game editors allow for the creation of game worlds, placement of characters and objects, scripting et cetera in an easy way. It is not uncommon that the game editor is included (in a modified form) with the videogame when it is released. This gives players the opportunity to build their own worlds to play in. A step further is a so-called mod(ification), which allows for the modification of the game specific code and sometimes even (parts of) the engine.

And now for the answer to the main question:

- What does it take to make a 3D videogame?
  The layered structure of a videogame tells how it is built up and thus what you need to know to make one. You will need to start out by getting (read: buying or building) a game engine. Furthermore, you will always need knowledge of 3D graphics, since in a game engine (renderer) you will always need to deal with it. Finally you must have knowledge of how a videogame works. Otherwise you cannot develop the scenario, gameplay, AI et cetera.

The above answer is of course not everything there is to making a videogame. Building videogames also takes a lot of experience. But when you look at making a game from a theoretical viewpoint as I did, I think that what I have described in this paper makes sense.

References

Angel, E. (2003). “Interactive Computer Graphics: A Top-Down Approach Using OpenGL™”, Third edition, ISBN 0-321-19044-0

Brownlee, J. (2002). “Finite State Machines (FSM)”, http://ai-depot.com/FiniteStateMachines/FSM.html

Howland, G. (2001). “The Basic Game Loop”, http://www.lupinegames.com/articles/gameloop.htm

Kelly, S.J. (2003). “Applying Artificial Intelligence, Search Algorithms and Neural Networks to Games”, http://www.generation5.org/content/2003/KellyMiniPaper.asp

Kuffner Jr., J. (2002). “Video Games (or the secret reason most Computer Science students started learning how to write code)”, http://www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15462/web.01f/notes/VideoGames.pdf

LaMothe, A. (1999). “Neural Netware”, http://www.gamedev.net/reference/articles/article771.asp

Matthews, J. (2000). “An Introduction to Game AI”, http://www.generation5.org/content/2000/app_game.asp

Patel, A. (2003). “Amit's Thoughts on Path-Finding and A-Star”, http://theory.stanford.edu/~amitp/GameProgramming/

Roy, P. (2002). “Direct3D vs. OpenGL: Which API to Use When, Where, and Why”, http://www.gamedev.net/reference/articles/article1775.asp

Sarle, W.S., ed. (1997). “Neural Network FAQ, part 1 of 7: Introduction, periodic posting to the Usenet newsgroup comp.ai.neural-nets”, ftp://ftp.sas.com/pub/neural/FAQ.html

Simpson, J. (2002). “Game Engine Anatomy 101”, http://www.extremetech.com/article2/0,1558,594,00.asp

Watt, A. & Policarpo, F. (2001). “3D Games: Real-time Rendering and Software Technology”, ISBN 0201-61921-0.

Weisstein, E.W. (2004). “Affine Transformation”, http://mathworld.wolfram.com/AffineTransformation.html (MathWorld--A Wolfram Web Resource)