

Boston College
Computer Science Department

Senior Thesis 2003
Evan McCarthy
A Three Dimensional Environment Suitable for Distribution over a Multicast Network
Prof. Robert Signorelli

0 Abstract

The purpose of this project is to integrate a three-dimensional model of a real-world environment into a distributed system based on multicast communication between nodes in the system. The development of three-dimensional environments (3DEs) revolutionized the computer graphics industry, in particular the gaming industry, by bringing computer graphics one step closer to replicating reality. The developed application uses a 3DE of a dormitory (specifically Ignacio Hall on the Boston College campus) to serve as a backdrop for a distributed system in which users compete against each other for resources and objectives (such as health powerups and to obtain a winning score). The development of a distributed mutual exclusion algorithm operating via multicast communication was necessary to ensure the proper performance of the developed distributed system. The created algorithm takes advantage of the characteristics of a multicast network and various synchronization mechanisms, such as a timestamp and the election from the multicast group of a central arbitrator. These synchronization safeguards not only enforce mutual exclusion, but also aid in the enforcement of reliable communication between nodes in the distributed system. The implementation of a dynamic rotating server selected from among the members of the multicast group, in combination with a timestamp applied to all messages sent between nodes in the system, forces the ordering of messages sent within the system. This adds a level of predictability and stability to developed system and its communication mechanism. The final implementation and integration of the 3DE, the distributed mutual exclusion algorithm and the reliable system based on the multicast communication

protocol result in a distributed, three-dimensional world, which mimics the everyday interaction between independent people vying for shared resources.

1 Introduction

The goal of this project is to create a realistic three-dimensional gaming environment suitable for distribution over a multicast network. The environment must resemble the model—in this case, Ignacio Hall—and must react with the players in a realistic way. For example, players should be able to climb stairs and open doors, but should not be allowed to walk through walls. From a visual perspective, realism will be added through the use of texturing and 3D design.

1.1 Game Engines

A game engine is framework that “integrates a 3D graphical gaming environment with intelligent techniques for generating and controlling interaction in order to create an engaging and coherent user experience.”¹ The controlling interaction refers to the heart of what a game engine does—it regulates the players in a way that produces a desired effect. If the point of a game is to simulate the life of a city over time, then the game engine should enforce rules that make the game realistic. Citizens should complain about taxes and city services should demand more funding. The engine should also enforce rules to make the game realistic—buildings should not float on water, for example. The game engine is the integration of the visual portion of the game with the logical portion of the game—it is the heart of the game itself.

¹ Young and Riedl, “Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds”

1.2 3D Graphics

In the early 1990s, id Software revolutionized the world of computer gaming with *Wolfenstein 3D*. The game used real-time ray casting to produce a 3D effect. The entities in the game were two-dimensional, but the depth was faked to give them the appearance of being three-dimensional. A few years later, the game *Quake* offered “full 3D entities that could move around in a fully polygonal 3D world with six degrees of freedom.”² Having six degrees of freedom means that an entity can move in the positive x , negative x , positive y , negative y , positive z , or negative z direction. The entity can also move in two, or even three, of these directions at one time. Thus, computer graphics had become fully 3D. Objects could yaw, pitch, and roll yet still look exactly as they would in the real world. The addition of a true third dimension to gaming opened a realm of possibilities. No longer would designers be confined to the xy -plane. Now, the z -axis was available.

1.3 OpenGL

This 3DE was created using OpenGL. OpenGL is a graphics application programming interface introduced by Silicon Graphics in 1992. Since that time, it has become the gaming industry’s most widely used graphics API. Many popular computer games have been written using OpenGL, including *Quake* and *Tribes 2*. OpenGL was chosen for this project for many reasons: its stability, reliability, good documentation, and portability make it a very attractive API.

2 Previous Work

² Hawkins and Astle, OpenGL Game Programming, p. 5

A good deal of previous work exists on gaming engines, although a large portion of it was developed for commercial use. After all, game engines are created to play games; and the computer gaming industry is a multibillion-dollar business.

One of the best known and perhaps most influential game engine projects is “Doom.” Created by ID Software in the early 1990s, Doom became one of the most successful video games in history. Its engine was one of the first 3D engines created. In fact, many of today’s games are built on top of Doom’s open source engine.

Other game engines are also available as open source. “Unreal Tournament” is one such engine. Young and Riedl investigated the creation of an interactive narrative on the Unreal Tournament engine in their 2003 work entitled “Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds.” In this project, they created a system that “creates a storyline at run time, allowing customization that takes into account a user’s ability, interests, previous experience and other contextual factors.”³ While not all of the content is relevant to this project, the paper does include some valuable information about what a game engine should do.

Legakis, Dorsey, and Gortler also provide interesting work with computer graphics. In August of 2001, they published “Feature-Based Cellular Texturing for Architectural Models,” a paper that explores using occupancy maps as a means to make textures appear more three-dimensional. They use a pattern-generating algorithm that matches the edges

³ Young and Riedl, “Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds”

of textures so they “align to achieve a particular aesthetic goal.”⁴ For example, bricks near the corner of a wall appear seamless, so as to give the appearance that the bricks actually go around the corner. Using this method when texturing supplies added realism to a 3DE.

Many game engines are available as open source, and while building a game on a pre-existing engine has its advantages, the flexibility needed for this project warranted the creation of a custom game engine. Because intimate data from the players and other objects in the 3DE need to be packaged, sent, and extracted with respect for size, it is necessary to have a strong understanding of the underlying implementation of the gaming engine. Also, because this project was also an exercise in modeling a real life three-dimensional environment, it was necessary to create the engine from scratch. The main reason for creating a unique engine, however, was the need to have graphics that easily interface with the networking portion of the project. It is important to have ability to change the implementation of an aspect of the game without needing to worry about the effect it would have on the code that I did not write. Because I created the gaming engine, I know exactly which pieces of information are needed to keep the game running accurately and realistically on multiple machines.

Further, because this portion of the project is also an exercise in modeling, it is not realistic to use a preexisting game engine. These engines have their limitation when it comes to designing a 3DE; the only way to get an accurate model of the target was build it all myself.

⁴ Legakis, Dorsey, and Gortler, “Feature-Based Cellular Texturing for Architectural Models”

3 Implementation

3.1 Modeling the Building

As one might expect, creating a 3D model of a large building is difficult. Fortunately, Ignacio Hall has many attributes that make it a desirable building to use as a model. First, with the exception of the bottom and top floors, all the floors are the same. This allowed me to create multiple models of the same floor and stack them on top of each other to give the appearance of a multi-story building. Second, Ignacio has a great deal of brickwork, which is an excellent pattern for texturing. Third, because the environment is inside rather than outside, there is no concern for the difficult task of mimicking nature. The appearance of sunlight and the horizon are much harder to create than the appearance of florescent lights and brick walls.

After acquiring a floor plan of the building, a rough estimate of the building's dimensions was needed. I determined approximately how close particular objects were to each other—space between doors, step height of staircases, etc.—and scaled to a size that would work well in a gaming environment. The scale came out to be around two game units per real foot. With floor plan and dimensions in hand drawing the virtual building was possible.

OpenGL allows the programmer to create simple polygons that can be used to form more complex object. A simple OpenGL command for drawing a square looks like this:

```
glBegin(GL_QUADS);
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
glEnd();
```

This code will draw a unit square, with a vertex on the origin, on the x -axis.

The quadrilateral is one type of polygon that OpenGL can draw. The most often used polygon, however, is the triangle. Even when drawing quadrilaterals, OpenGL programs will break the quad up into two triangles instead of keeping it as a whole shape because drawing two triangles is actually faster than drawing one quad. This is because “most 3D accelerated hardware is highly optimized for the drawing of triangles. In fact, you will see many 3D benchmarks measure in triangles per second.”⁵ To speed up the rendering of the geometric shapes that comprise the building, the polygons were broken down into a collection of triangles. For the above-mentioned unit square, the triangle rendering approach would look like this:

```
glBegin(GL_TRIANGLES);
    //start of first triangle
    glVertex3f(0.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 1.0f, 0.0f);

    //start of second triangle
    glVertex3f(1.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 0.0f, 0.0f);
    glVertex3f(0.0f, 0.0f, 0.0f);
glEnd();
```

⁵ Wright and Sweet, OpenGL Super Bible, p. 146.

Counting walls, floors, ceilings, doorframes, and stairs, each floor of the building is made up of approximately 1300 triangles. When multiple floors are stacked to create the building, the number of triangles grows quickly. As a result, any time saved by drawing triangles instead of quads is beneficial.

Using `GL_TRIANGLES` is only one method used to speed up the drawing of the 3DE. Another technique is culling. Culling is based on the principle that when you look at a geometric shape, you can see only one side at a time. Therefore, if you can only see the front, why bother drawing the back? In OpenGL, “both sides of a polygon are drawn separately.”⁶ If only one side of the polygon is viewable, it is not necessary to draw the other side. Culling allows the programmer to specify which side of a polygon is drawn, while the other side is not rendered. This method is incredibly helpful when drawing a large number of polygons that are only viewable from one side. A good subject for culling would be a 3-D wall. The two outside faces of the wall should be drawn, however, the user never sees the two inside faces of the wall. Culling out these two faces reduces the number of polygons drawn by half and results in significant improvement in speed and efficiency when a large number of polygons are drawn. To use culling, a programmer must first enable it and then specify in which direction he will list the vertices of his shapes. If he chooses clockwise—as I did—he will need to list all vertices in a clockwise order. If he lists them in a counterclockwise order, he will not see the shape unless he looks at it from the other side.

⁶ Hawkins, Kevin and Dave Astle, OpenGL Game Programming, p. 106

Yet another method for reducing rendering time is the use of display lists. A display list allows you to “cache commands if you plan to redraw the same geometry multiple times.”⁷ Display lists “provide a means to precompile commonly used sets of commands, allowing them to be used at a lower cost.”⁸ Basically, when a display list is used, the information pertaining to the drawing of a certain object is stored in the list as preprocessed commands. The next time that the object needs to be drawn, the necessary commands will be pulled out of the display list instead of being reprocessed. This works well for objects that never change state because reprocessing will never be necessary. Because the building is a static collection of geometric shapes, it is a perfect candidate for a display list. Functions that are frequently called but never change state should be placed in display lists so that calculations only take place once, rather than every time that portion of the code executes. “Display lists are faster than just reexecuting the same OpenGL functions singly.”⁹ Also, the results of any mathematical calculations are stored and passed to OpenGL functions as needed. A significant amount of optimization occurs by using displaying lists; in fact, in OpenGL, a 20% increase in performance can quickly be obtained by using display lists, especially when the number of polygons being drawn is large, as is the case in our 3DE.

Below is the floor plan that I used to model Ignacio Hall.

⁷ Woo, Neider, and Davis, OpenGL Programming Guide, p.252

⁸ Hawkins, Kevin and Dave Astle, OpenGL Game Programming, p. 320

⁹Wright and Sweet, OpenGL Super Bible, p. 334

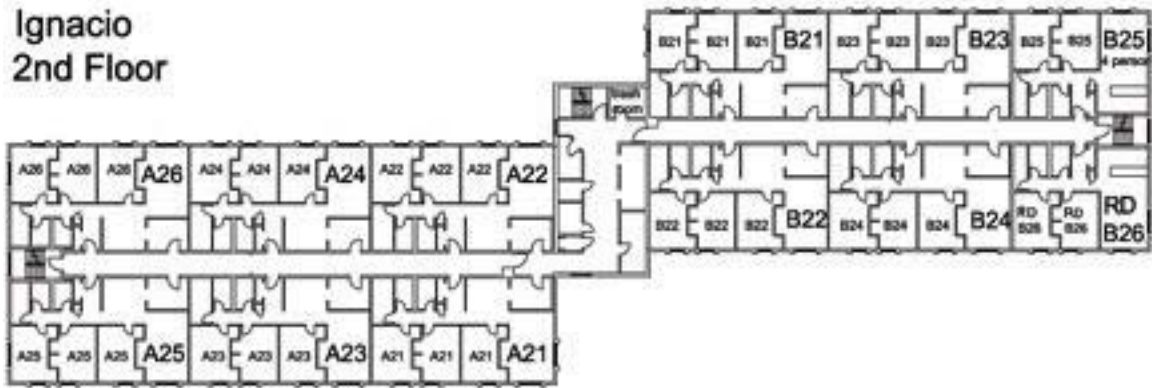


Figure 0: Ignacio Hall Floor Plan
courtesy of the Boston College Office of Residential Life

I constructed the floor in three distinct sections: the left wing, the center section, and the right wing. The left wing consists of a hallway, a staircase, and six suites. The suites themselves are made up of ten identical rooms. The center section is made up of the hallway, a staircase, two elevators, and two rooms. The three rooms on the left side of the center section hallway are not accessible in the game. The left wing and the right wing are identical in the game, except that the right wing is rotated 180° and translated up and to the right from the left wing. Because they are the same, to create the right wing, I simply copied the left wing and preformed the appropriate translations.

Once all the appropriate geometric shapes were assembled, I added textures to give the 3DE a more realistic appearance. Texture mapping is “the process of applying a texture image to a surface to produce patterned surfaces such as wood or marble.”¹⁰ Texture mapping can be a complicated process. The first step is to decide what size you want your texture images to be. If the images are too large, the graphics will be bogged down. If the images are too small, the textures will be of poor quality. I chose to use 32x32 pixel texture images. These images are then read into an image array that will be used by OpenGL to texture surfaces. Portable Pixel Maps (ppm) files are well suited for this because of their format. PPM files are simply text files that represent colors in RGB triplets. The RGB values are relative to the maximum color value, which is defined near the beginning of the PPM file. I used an algorithm from William Ames (Boston College) to transfer the data from the images to the image array. The algorithm reads in the first triplet of RGB values then divides each by the maximum color value. This will produce a value between zero and one, which is how OpenGL represents color. To use the textures, the programmer must first generate a list of textures so that OpenGL can reference them by index. Each image array is bound to a spot in the texture list. Once this is done, the textures are ready for use. To do so, the programmer must first call up the desired texture from the texture list and bind it to the current drawing target. Once this is done, the programmer must specify the *s* and *t* texture coefficients. The texture coefficients are repetition ratios—that is, they represent the number of times to repeat a texture horizontally and vertically over a given surface. The coefficients are represented by arrays that specify the *x*, *y*, and *z* coefficients for a given direction. Now that the

¹⁰ Wright and Sweet, OpenFL Super Bible, p.700

textures are bound and their coefficients are specified, all geometric shapes that are drawn will be textured by the given image. If the programmer draws a shape on the xy -plane and then wants to draw another in the yz -plane, he must specify the new coefficients for the shape. This type of texture mapping is known as automatic texture generation because the programmer only needs to specify the number of repetitions he wants and then allows OpenGL to calculate the exact texture coordinates for each image.

3.2 Creating the First Person Perspective

The first person perspective (FPP) simulates the feeling that the user is viewing the 3DE through his own eyes. The FPP is created using a combination of the `gluPerspective` function, the `gluLookAt` function, and trigonometry.

The `gluPerspective` function takes four parameters: the field of view angle, aspect ratio, near clipping plane, and the far clipping plane. The field of view angle is simply the angle through which the user can see in the y -direction. It defines the vertical vision of the user. The larger the field of view angle, the more the user can see above him and below him.



Figure 1: Example of a small field of view angle

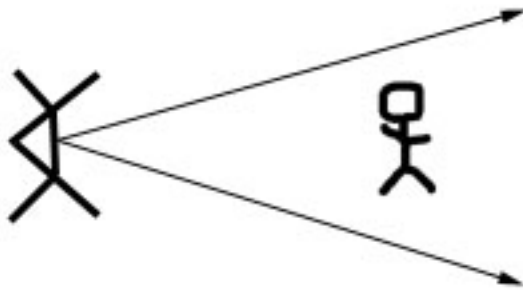


Figure 2: Example of a large field of view angle

The aspect ratio determines the field of view in the x direction. It is the ratio of the width to the height. As the ratio grows, the peripheral vision of the player becomes smaller because more x values are packed into the horizontal view.



Figure 3: Example of a small aspect ratio



Figure 4: Example of a large aspect ratio

The near and far clipping planes specify points where shapes are not drawn. The near clipping plane is usually specified at a point close to zero while the far clipping plane can be specified at any point out into the scene. Any point that does not fall between the two clipping planes is not rendered. This feature can be used to save rendering time. Points that are far off in the distance, or points that are behind the viewer do not need to be drawn. By cutting these points out with clipping planes, the programmer saves processor time.

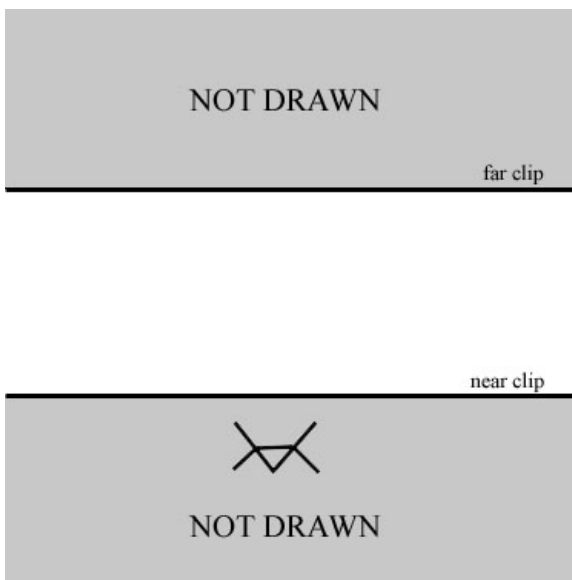


Figure 5: The near and far clipping planes

Using the `gluLookAt` function is a bit more complicated. The `gluLookAt` function takes nine parameters; the first three specify the location of the eye; the second three specify a reference or “look-at” point; the last three specify the “up” vector.

Determining the eye location and the up vector are easy enough. The eye location is simply the location of the player. The up vector is $(0, 1, 0)$ because in our 3DE positive y is the up direction. Finding the look-at point requires some trigonometry. Because our 3DE is truly three dimensional, players can look in look not only side to side (yaw) but also up and down (pitch).

The yaw is dependant on the sine and cosine of the player’s horizontal angle. In our 3DE, a player with a horizontal angle of 0 is looking down the negative x -axis. To calculate the look-at point for a player, let us look at some basic trigonometry.

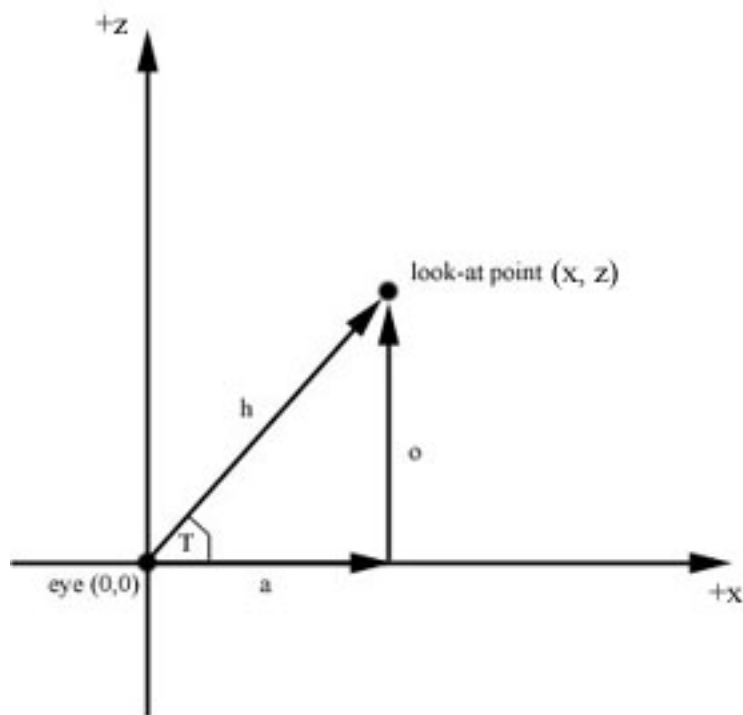


Figure 6: Yaw trigonometry

The hypotenuse (h) of the triangle represents how far away from the player the look-at point is. The x -coordinate of the look-at point is equal to the adjacent side (a) of the above triangle. The adjacent side of a triangle is equal to the cosine of the angle, therefore, $x = a = \cos(T) \times h$. The z -coordinate is equal to the opposite side (o) of the triangle, which is equal to the sine of the angle. Therefore, $z = o = \sin(T) \times h$. Using these two coordinates to change to look-at point, the direction in which the player looks at horizontally can be changed.

The player's pitch is dependant upon the tangent of the player's vertical angle. In our 3DE, a player with a vertical angle of 0 is looking parallel to the xz -plane. Again, basic trigonometry can be used to determine the new look-at point.

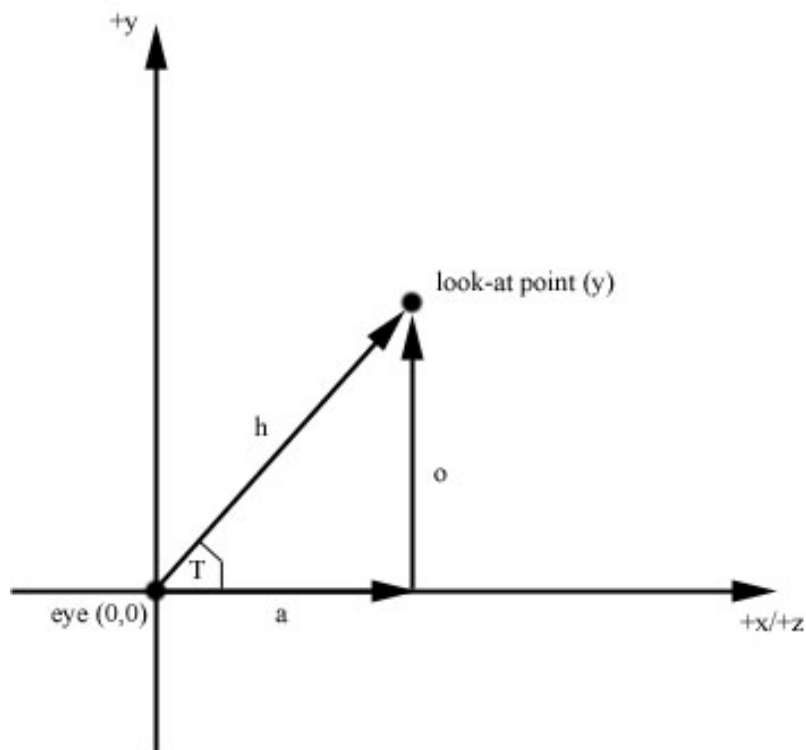


Figure 7: Pitch trigonometry

Because we already have determined the x - and z -coordinates for the look-at point, the adjacent side of the triangle (a) is equal to $\sqrt{x^2 + z^2}$. The y -coordinate of the look-at point is equal to the opposite side of the triangle (o). Because the $\tan(T) = o/a$, we find that $y = o = \tan(T) \times a$.

We now have our look-at point. Every time the player is drawn, the look-at point will be offset from the player's eye point by $(x, y, z) = (\cos(T) \times h, \tan(T) \times h, \sin(T) \times h)$ where h is some constant. As the player's angles change, so will the look-at point because it is dependent on these angles. This will create the effect that the user is moving his head around and looking in different directions.

3.3 Modeling Player Movement

Modeling the movement of players involves defining the way in which the players move through space (physical movement) as well as the way they look while doing so (animation). The two types of movement must be tied together so that when observed by other players in the game, each player's motion is realistic.

The physical movement of the players allows them to navigate through the 3DE. Each player can perform five motions: movement in the forward direction, movement in the backward direction, strafe left, strafe right, and squatting. The first four motions involve a change on the xz -plane while the squatting motion involves a change in the y -direction.

The forward and backward movement, much like the previously describe calculation of the look-at point, rely heavily on trigonometry. The change in x is equal to a constant $STEP_SIZE \times \cos(T)$ where T = the player's horizontal direction. The change in z is equal to $STEP_SIZE \times \sin(T)$. The player's y -value is not dependent on movement in the xz -plane and therefore stays constant here. The only time a player's y -value will change due to movement in the xz -plane is when the player encounters a set of stairs. If a player crosses the face of a step, his y -value is incremented or decremented an appropriate amount depending on whether they are traveling up the stairs or down the stairs. Other than this case, a player's y -value is not changed due to movement in the xz -plane.

The strafing motion for the players extends from the forward/backwards motion. A strafe is a sidestep motion perpendicular to the players line of sight which results in the player maintaining a new line of sight parallel to the original line of sight.

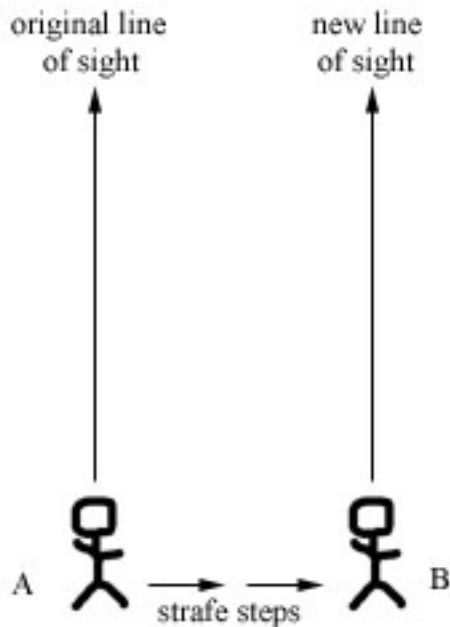


Figure 8: Strafing

Notice that the strafe steps are perpendicular to the player's line of sight. At position *B*, the player's new line of sight is parallel to the line of sight from position *A*. The player truly is "side stepping."

Here is the algorithm used to strafe the players:

```
rotate player +90°;
step player forward;
rotate player -90°;
```

The algorithm to achieve the strafe is fairly simple considering how difficult it would be to calculate a point `STEP_SIZE` units to the right or left of the player without rotating the player at all. This would involve extending the side of the triangle by the desired amount and recalculating the new point. The rotation method allows for reuse

of code and reduces the complexity of the trigonometry, so I elected to use it to create the strafe motion.

The squatting motion is a very basic motion that simply lowers the player's eye level by a predetermined constant. While squatting, the player observes the 3DE from a slightly different y -value. The player's y -value is reset when they come out of the squatting position.

The animation of the player's movements is important to make the game realistic for other players in the game. Each animated motion is comprised of three frames: one based on the movement of the right leg, one based on the movement of the left leg, and one with no leg movement. When a user instructs his player to take a step, the player is drawn using the frame for right leg motion. A flag is then set indicating that the next time the player steps, the left leg should lead. This alternating of the legs creates the illusion that the player is stepping forward with one leg then the other. Originally, I had hoped to have several frames of animation for each step (that is having frames that rendered the intermediate phases of taking a step), however, while testing this implementation, I found that the players would finish their actual movement (that is movement from point a to point b) before the animation was complete. The reason for this is because a player's current animation frame is incremented to the next frame when the player's draw function is called, which is called by the display method. However, a change in a player's location forces display to be recalled so that the graphics stay in sync with the game state. Therefore, a

player will request a move, recall display and be redrawn in the new spot. The animation, however, will have just begun and will not be complete until n calls of display (where n is the number of frames in the animation). This means that the movement will be done on the first display call, but the animation will continue for another $n-1$ calls of the display function. This causes the players' legs to continue to move after the body has come to a stop. To obtain synchronization between the players' positional movement and the animation of the legs, the intermediate frames were scrapped in favor of the single-frame, alternating foot implementation. Below is a frame-by-frame view of the animation for a forward step.

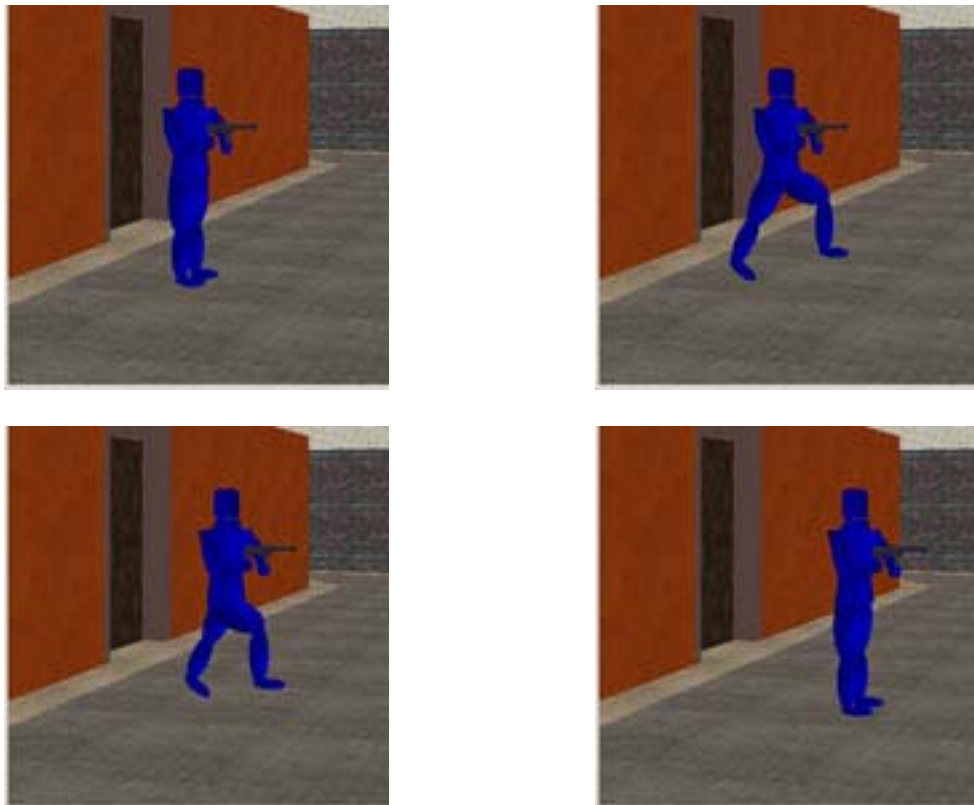


Figure 9: Forward step animation

While the addition of intermediate frames would have made the animation smoother, the chances of the animation looking choppy once the game engine is distributed is greater. Because the current frame number of each player must be passed to all the other players, drawing intermediate frames could be a problem if packets are lost or if there is a bottleneck in the transfer. Lost packets would result in the intermediate frames being skipped, which would make the animation choppy. Packet backlog would result in slow animation during the backup and fast animation once all the packets were received. Therefore, for simplicity's sake, the intermediate frames were cut from the animation.

3.4 Creating Game State

We have an arena in which to play, we have players who can move, but one important element of any game has yet to be addressed: What is the objective? This game engine is built to model a laser tag game, so obviously the objective is to navigate the 3DE and score points by “tagging” other players with your laser gun. Clearly, there must be some kind of game state that keeps track of health, hits, and score.

Each player is given an amount of health when created. As they encounter other players, they can either shoot those players or be shot by those players. If a player takes a hit, his health is decremented an appropriate amount. A user can keep track of his health by looking at the health bar in the upper left hand corner of his screen. Players can recharge their health by picking up health packs that are distributed throughout the level. If a player's health reaches zero, that player is “dead” and cannot continue playing until he respawns in his team's base.

The game engine regulates when a laser hits a player. When a player shoots a laser, the laser travels in a straight path. If at anytime that laser comes within the defined boundaries of player p , then p has been hit. P 's health will be decremented. In an effort to minimize network traffic, reduce game screen clutter, and more closely mimic an actual laser tag game, each player can only fire one laser beam at a time. While a laser has been fired, a flag is set in the player that indicates that the player cannot fire again until the recharge time has passed. The recharge time is a counter that is incremented every time the player's draw function is called. As a consequence of this, users with faster machines have an advantage because their counters will expire more quickly than those using slower machines. This advantage is somewhat unfair, but is not uncommon in the gaming industry. Those using sub par machines have to accept the fact that their machines cannot keep up with the higher-powered computers.

3.5 Edge Detection

In order for the walls in the 3DE to have meaning, they have to impede player movement. To this end, edge detection must occur. Objects should not be allowed to base through walls, floors, and ceilings. When a player runs into a wall, the game engine tries to adjust the player's trajectory so that the player slides along the wall rather than getting stuck on it. The adjustment is made by checking where the player would go if his angle was slightly changed. If that point is inbounds, the player moves there. If it is not, the engine changes the angle again. If no satisfactory point

is found within three iterations of the algorithm, then the player is not allowed to move at all. The basic algorithm for edge detection in our 3DE looks like this:

```
checkpoint(point p){
    counter = 0;
    if nextPoint is inbounds and
        nextPoint does not hit wall and
        nextPoint does not hit another player{
        allow movement;
        return;
    }
    else{
        while counter < 3{
            adjust angle;
            recalculate nextPoint;
            if nextPoint is inbounds and
                nextPoint does not hit wall and
                nextPoint does not hit another player{
                allow movement;
                return;
            }
            else{
                increment counter;
            }
        }
    }

    disallow movement;
    return;
}
```

There are two types of edge detection that take place in our 3DE. The first, which I shall call line edge detection, detects movement across a particular line. The second, which I shall call block edge detection, detects movement into a predefined area of space.

The line edge detection is used when it is necessary to know which direction the object is traveling. This is exemplified on the staircases. If there exists a staircase S such that the edge of a stair s_1 is at $x = 30$ and the edge of another stair s_2 is at $x = 31$, then it is necessary to adjust the player's y value as he crosses the line $x = 30$. But in which direction should it be adjusted? If the player is coming down from s_2 onto s_1 then the y -value should be lowered. If the player is moving from s_1 to s_2 then the y -value should be increased. The best way to determine which direction the player is traveling is to compare the player's current point to the player's next point. In the case of S , if the player's current x -value is less than the player's next x -value, then the player is moving up the stairs. The y -value should be increased. If the opposite is true—that is, the player's current x -value is greater than the next x -value—then the player is going down the stairs and the y -value should be decreased. Clearly, in this case, it is important to know what the player's direction is.

If the player's direction is not important then block edge direction is the right course of action. Block edge detection does not take the player's direction into account; rather it concerns itself only with the player's position. This type of edge detection should be used on objects such as walls and other players. With block edge detection, the programmer “blocks” out areas into which players cannot travel. For example, if a wall occupies the space enclosed by the lines $x = 0$, $x = 10$, $z = 0$, and $z = 10$, then if a player ever requests to move into that space, the game engine will not allow it. Line edge detection can also be used here, but with at least twice the cost. If it were used,

the engine would have to check both sides of the wall instead of just the area inside of the wall. Clearly, block edge detection is the right course of action when dealing with walls and objects.

3.6 Making It Distributed

As stated earlier, the goal of this project is to create a realistic three-dimensional gaming environment suitable for distribution over a multicast network. In a multicast network, users are not concerned with whom they are communicating; rather they simply send and receive information to a group. To this end, the player objects must contain information that is easily packaged and easily extracted from a packet. Other objects in the game must send information to other users as well. The visibility and availability of health packs must be relayed to other users; and elevator doors must be made to open and close in a distributed fashion so everyone in the game can see when a player interacts with a door.

The distribution of the information itself is beyond the scope of my portion of the project. Please see work by Frank Mazzacano and Jonathan Pearlin for more information on the distribution of data. However, my responsibilities did include creating the objects in an easily distributable way.

Every user has a global PlayerList object that contains pointers to every player in the game. When the player updates his states (let update in state mean moving, shooting, etc.) he sends his new information to the multicast group. When a player receives

new information about another player from the group, he updates the Player object in his PlayerList accordingly. Each client machine is responsible for drawing their own version of the players in the game; the trick is to make all the versions match each other.

Likewise, each player has a list of ElevatorDoors and a list of HealthPacks that must be updated to match the current state of the game. If a player picks up a health pack or opens a door, he sends the appropriate information to the group. Other members of the group will then extract this new state information and update their representation of the world appropriately.

As stated earlier, every client in the multicast group is responsible for interpreting and drawing the information they have about the state of the game as a whole. If clients have different state information, they will not display the same game state—for example, a player may appear at point *a* on one client machine, but at point *b* on another. While the synchronization of distribution is not dealt with in this portion of the project, it is important to be aware of it. Again, see work by Frank Mazzacano and Jonathan Pearlin for more information of the distribution of game state information.

4 Conclusion

On a whole, I would rate this project as a success. Personally, I achieved my goal of creating a distributable and realistic three-dimensional gaming environment. While our

game is a far cry from the latest state-of-the-art video games, it certainly is a step in the right direction. In fact, I would say that it is comparable to *Goldeneye*, release for the Nintendo 64 gaming system in 1998. That game took several years and hundreds of programmers to develop; we created ours in under a year.

I consider many parts of my portion of the project to be successful. In particular, I am rather proud of how well the model of Ignacio Hall turned out. With the exception of a few small details, the model is an exact replica of the actual building. I credit this to the realization that indoor scenes are much easier to model than outdoor scenes. After spending a good deal of time working to create an outdoor city scene for the game engine, I realized that the switch to an indoor scene would be wise. A city scene poses too many problems, including modeling sunlight and the horizon. Another advantage of choosing a building like Ignacio is the amount of repetition found. All the floors are the same, so once you have created one, you have created an infinite number of them. This experience has truly shown me the value of reusing code and planning so that code can be reusing.

I am also impressed with the speed of the graphics. Although I have no numbers on the actual frame rate of the graphics, the movement seems to be fairly smooth to the human eye. Somewhat surprising to me is that the graphics are slower when stand-alone than they are when distributed; although I suspect this is because the program is getting caught up looking for the networking portion and thus ends up running slower. This, of course, is not devastating to the cause; after all, the game engine is supposed to be distributed.

The animation of the players turned out better than I thought it would. As shown in Section 3.3, the players are animated whenever they take a step forward, backward, or to the side. During the implementation process, I was afraid that cutting out intermediate frames would result in sub par animation. As it turns out, the opposite was true. The animation looks much better without the intermediate frames that it did with them. The removal of these frames also simplified the code and the game state distribution process. This has instilled in me two thoughts. The first: less truly is more. The second: animation, especially video game animation, is all smoke and mirrors. My implementation of the animation does not come anywhere close to showing all the “key” positions of a leg in motion. It does, however, create the illusion of fluid motion. Illusion, often times, is just as good—if not better—than reality.

Two aspects of the game engine did not turn out as well as I planned. The representations of the players are not spectacular and the implementation of the lasers needs more work.

The players are not works of art. It would have been nice if I could have created players with realistic faces and features. To do this, I could have had to use some type of mesh mapping. A mesh is simply a “collection of polygons, along with a normal vector associated with each vertex of each polygon.”¹¹ Meshes can be used to model complex surfaces. If the polygons in the mesh are small and the transitions between neighboring polygons is subtle, the mesh will appear to be quite smooth. A mesh such as this would

¹¹ Hill, F.S., Computer Graphics Using OpenGL, p.290

require a large number of very small polygons. Unfortunately, calculating the vertices for hundreds of polygons did not fit into our timetable. Given two more months on this project and I am confident I could have created some great looking players. I have a new found appreciation for those who model the human body. It truly is the most complicated structure out there. Anyone who can model it well deserves a lot of credit.

Another aspect of the game engine that fell short of my expectations is the implementation of the lasers. I tried to add crosshairs to the game engine so that users can see exactly where they are shooting, however, it proved to be difficult because of the way the lasers work. Because the lasers are shot from a point that is slightly lower and to the right of the players eye, it is impossible for the player to actually observe exactly where the laser is going. The problem is parallax. If the player is close to an object, the laser will hit lower and right of where the user thinks it should. If the player is far from an object, the laser may travel over the top of the target. Crosshairs could be added if the player performed some form of depth testing—that is determined the closest object that fell within the crosshairs—and then targeted his laser beam at that point. This way, the laser would always travel exactly where the player wanted it to go. This reimplementation of the laser beams would require quite a bit of time; again, with a few more months, I'm sure I could create something more satisfactory.

Again, I believe the project as a whole to be a success. The three dimensional environment is distributed and players can interact with each other. Although the

appearance of the players themselves is not entirely realistic, their movements look good, as does the world surrounding them.

I would like to thank Frank Mazzacano and Jonathan Pearlin for their work on this project. The 3DE would never have become distributed were it not for their networking knowledge. I would also like to thank all those who helped to test our game. Their time and input expedited the debugging process. William Ames also must be recognized for helping me become more familiar with OpenGL, both in and out of class. Finally, I would like to thank Robert Signorile for his work as our thesis advisor. He gave us the flexibility to explore but at the same time kept us on schedule. Thank you.

Sources Cited

Hawkins, Kevin, and Dave Astle, OpenGL Game Programming, Prima Publishing, 2001.

Hill, F.S., Computer Graphics Using OpenGL, Prentice Hall, 2001.

Legakis, Justin, Julie Dorsey, and Steven Gortler, "Feature-Based Cellular Texturing for Architectural Models," August 2001.

Woo, Mason, Jackie Neider, and Tom Davis, OpenGL Programming Guide, Addison-Wesley Developers Press, 1997.

Wright, Richard, and Michael Sweet, OpenGL Super Bible, Waite Press Group, 1996.

Young, Richard, and Mark Riedl, "Towards an Architecture for Intelligent Control of Narrative in Interactive Virtual Worlds," Liquid Narrative Group, North Carolina State University, January 2003.