Zombs

Project Write-Up

By Evan Kleist

June 7th, 2011

Table of Contents

1. Introduction

With the recent boom in the video game industry, the design and development of games has become an extremely competitive and sought-after job market for software developers. In 2009 the video game industry generated over 19.6 billion dollars. This overwhelming revenue surpassed both music and movies. With this large market, hardware is constantly being optimized to handle the growing needs of 'cutting-edge' video game systems.

A modern computer is capable of performing billions of computations per second. However, video games are made to represent entire worlds and sometimes can be expected to represent large sets of data that go around. Performing operations without concern for efficiency can easily bog down a computer's processor and create an unplayable game. This makes it important to use similar design and optimization techniques used in other forms of more traditional programming applications.

The *Zombs* project was an appealing choice for our group because of the present popularity and demand for video games, and the opportunity to put to practice many essential elements of Computer Science such as design, prototyping, teamwork, implementation, and testing. Zombies have been increasingly appearing in all forms of popular culture. Regularly depicted in horror and fantasy based entertainment, Zombies have captured the interest of millions of people world wide.

Using zombies as the primary theme, the *Zombs* project was initially conceived as a proposal in our CPE476++, Real-Time 3D Computer Graphics Software Systems class. The project began as a two quarter process with the team being composed of five members, but due to the lack of contributions from one team member, we only had four members for the second quarter (Alan DeLonga, Evan Kleist, Jordan Gasch, Reece Engle). The initial game idea was a mix between bomber-man and Diablo II. As the project progressed, we chose to follow a more ominous ambiance with our models, lighting, camera angle and background sounds. By having a democratic atmosphere where all members gave input, our project became a cohesive culmination of all of our

ideas. The final result differs quite dramatically from the initial proposal, but the team is pleased with the outcome.

2. Project Overview

The basic story behind Zombs is you are a survivor in a zombie apocalypse. You wake up in a hospital after being in an car accident, and your main objective is to find your wife who was also in the car. Your walking is labored and you can only sprint for short distances. As you progress through the level, you realize something has gone horribly wrong and that zombies have over run the hospital. You navigate the levels from a 3rd person perspective and utilize various weapons to fight off the zombie hordes.

Since the setting is the inside of a hospital, Zombs uses hospital style models while preserving some darkness since it is the zombie apocalypse. Throughout the level you will encounter patient rooms with complex models for beds, patients, shrubbery, waiting chairs, and other items.

The player navigates through the level by following objectives. The objectives are displayed on the HUD, and change as you progress through the level. In the first level, the basic game controls are explained via the objective system. The basic controls are 'WASD' to move, left click to melee attack, and 'SPACE' bar for special weapons. Special weapons which have been aquired are cycled with 'TAB'. The camera rotation is bound to mouse movement, so moving the mouse at any point will rotate the camera about the character.

Zombs was written in C++ and used OpenGL as the graphics library. The windowing system being used is SDL. For sound, Zombs uses an extension of SDL called SDL_mixer. Rendering of text is done by another extension of SDL called SDL_ttf. The MD2 model loader is built upon the GLEW library.

3. Related Works

The main inspiration for Zombs was Blizzard Entertainments Diablo 2. We used a very similar camera as Diablo 2, and tried for the same “dark” look and feel of it. At one phase of implementation, which is no longer in the release, we also had walls becoming transparent if they obstructed the view of the player.

Blizzard Entertainment's Diablo 2

In the later phases of implementation we drifted from Diablo 2 and went more like Capcom's Resident Evil 2. The main adjustment that changed the look and feel from Diablo 2 to Resident Evil 2 was dropping the camera down further into the level so the camera would no longer see over the tops of walls. With this change, it was no longer necessary to have transparent walls.

Capcom's Resident Evil 2

4. Algorithms Overview

In order to produce an exciting game in a 3D environment while maintaining optimal performance, many technologies and algorithms were included in Zombs. While some were required technologies for class curriculum, others were added by choice. This paper will only go into detail for the particular algorithms implemented by myself. For details on the other algorithms, see the senior project writeup for the team member indicated in parentheses. While this list is not exhaustive of all code in the project, it hits on all of the harder components.

- Real Time Movement/Update (All)
- View Frustum Culling (Evan Kleist)
- Particle Generation (Alan DeLonga)
- Spatial Data Structures (Evan Kleist)

- Per Pixel Shading (Jordan Gasch)
- Collision Detection (Evan Kleist)
- Collision Response (Evan Kleist)
- Model Loading/Creation (Alan DeLonga)
- Animation (Alan Delonga)
- Artificial Intelligence (Jordan Gasch)
- Shadows (Jordan Gasch)
- Sounds (Alan DeLonga, Evan Kleist)
- Level Editor (Reece Engle)
- Smart Camera (Evan Kleist)
- Bombs (Projectile System) (Evan Kleist)
- Heads-Up-Display (Alan DeLonga, Reece Engle, Evan Kleist)
- Inventory (Alan DeLonga)
- Transparency (Evan Kleist)

5. Algorithms Detail

The following subsections go into technical detail of the algorithms personally implemented.

5A. Collision Detection

The collision detection system consists of Axis-Aligned Bounding Boxes and Bounding Spheres. Every object in the world has one of these bounding volumes. There are then collision detection algorithms for sphere-sphere, box-sphere, sphere-box, and box-box in order to determine

whether the bounding volumes intersect or not. In addition to bounding volume collision tests, it was also necessary to implement two additional collision-related tests, line-sphere and line-box. These would test whether a given line segment intersected a bounding volume. which was primarily used in detecting whether the camera's view of the player was obstructed.

Collision detection is an integral problem to all video games. How do you computationally detect the intersection of two objects in the world? While doing so, you must also keep in mind performance. A true pixel collision detection system would require so much computation that it would not be realistic for a real-time video game. Because of this, the problem needs to be generalized and simplified in order to keep the performance high.

Zombs uses a combination of Axis-Aligned Bounding Boxes(AABB) and Bounding Spheres(BS). A bounding box is a box that completely contains the given object. That is, it is the smallest possible box that can fit the object. An AABB is a bounding box where the edges of the box only travel in one dimension. This constraint results in much easier computations at the cost of bounding boxes that may not be the best fit. A bounding sphere is similar to a bounding box in that it completely contains the given object. Zombs uses a combination of these two bounding objects as some objects are better contained in spheres (a bomb, for example) while some are better contained in boxes (a bed, for example.)

The implementation begins with the "interface" BoundingObject. A BoundingObject is either a BoundingBox or a BoundingSphere. Every BoundingObject has a position, which is refers to the center of the object. A BoundingBox also has three dimensions (the size in the x, y, and z dimension) while a BoundingSphere has a radius. Every object has a corresponding BoundingObject. A BoundingObject then has the following methods:

void draw()

This was a straightforward method used for debugging that would draw the corresponding BoundingObject

bool collide(BoundingObject other)

This is the heart of collision detection. It first tests whether *other* is a BoundingSphere or BoundingBox. It needs this information in order to determine whether we are doing sphere-sphere, sphere-box, box-sphere, or box-box collision.

Sphere-Sphere collision is tested by calculating the distance between the center of the two spheres. If this distance is smaller than the combined radius then the spheres collided, otherwise they did not.

Sphere-Box and Box-Sphere collision are calculated by first finding the closest point on the box to the spheres center. This is done by clamping the spheres center (in x, y, and z) to the boxes edges. It then computes the distance from the center of the sphere to this clamped position. If this distance is smaller than the radius they collided, otherwise they did not.

Box-Box collision is calculated by first finding the min and max coordinates of each bounding box, in all three dimensions. It is than a simple interval test to see if the boxes overlap.

Point getBiggestMovement(Point oldP, Point newP, float radius)

This function was used in collision response, in order to “slide” along walls or other objects. Given a current position, *oldP*, and a position you want to move to, *newP,* and the players BoundingSphere radius, it determines whether or not you can move to that new position. If you can't, it returns the furthest position you can move to along the segment created by oldP and newP without colliding.

bool contains(Point p)

This is a straightforward method that determines whether or not a given position is contained by the BoundingObject. It is essentially a simplified version of the collide() method, and uses the same algorithms described above.

float distanceFromPoint(Point p)

This method determines the distance from the bounding object to the point *p.* The distance returned isn't the distance from p to the center, but rather p to the edge of the BoundingObject. For a sphere, it takes the distance from p to the center and then adds the radius. For a box, it first finds the closest point on the edge of the box (same algorithm as collide()) and then returns the distance from p to that point.

bool lineIntersect(Point p1, Point p2)

This method determines whether a given line segment collides with the BoundingObject. It was used in determining whether the cameras view of the player was blocked by an object.

For a sphere, it starts out by computing two direction vectors, d and f. D is the direction vector of the line segment. F is the direction vector from p1 to the spheres center. To test for intersection, it uses the discriminant of the quadratic equation. A is d.dot(d), which is the squared magnitude of the line segment. B is 2 * f.dot(d), where f.dot(d) represents how much the vectors are pointing in the same direction. The key here is that is the line segment intersects, the direction of the line segment better be close to the direction towards the sphere. C is f.dot(f) – r * r, which is the squared magnitude of the sphere direction minus the radius squared. It then computes the discriminant of the quadratic equation. If the discriminant is negative, then no intersection exists. If the discriminant

is positive, then an intersection exists.

5B. Collision Response

Along with collision detection comes collision response. Once a collision has been detected using the algorithms in the previous section, how do we want the objects to react? Specifically, collision response in the change in motion when two objects collide. For the first phase of our game, the collision response was quite simple: stop movement. While this approach gave us the proper collision detection/response so objects would not pass through each other, it got quite frustrating because if you collided with an object you would completely stop, regardless of the angle of collision. What we achieved for 200% was a less frustrating approach, but still not physics based or truly realistic. If movement causes collision, it takes your current position and the position you are trying to move to, and calls getBiggestMovement(oldP, newP) which returns the best position you can move to without colliding. This lets you slide along objects if you are not running straight into them, making movement much easier.

5C. Smart Camera

The camera in Zombs is a third person aerial camera. For the first phase of the game, it was much like the camera in Diablo. It sat high in the sky and looked down at the player at about a 45 degree angle. This gave the player a "godlike" position as they could see over the tops of walls and into adjacent rooms. This however was not what we really want. In order to give a much scarier feel to the game, we dropped the camera down into the world so the player could no longer see over the tops of walls. This brought about the problem of the view of the character being obstructed by walls or other objects. In order to prevent this, the distance of the camera from the player would increase/decrease in

order to always keep the player in sight of the camera.

The algorithm for this started with camera movement. Every time the camera moved, it was necessary to check for visibility of the character. To test for obstruction the camera would use lineIntersection() where the line being tested was generated from the characters position and the cameras position. If lineIntersection() came back true, we know that an object stands between the character and the camera and the camera needed to zoom in in order to restore visibility. The camera then finds the furthest point along the segment such that the obstruction is avoided.

5D. Bombs(Projectile System)

The only projectile weapon existing in the game is the ability to throw a bomb. Aiming is broken down into two parts, distance and direction. The direction is a straightforward calculation of the cameras forward vector, as the bomb is always thrown directly in front of the player. The distance is calculated by the amount of time the space bar is held down. The distance oscillates on a three second period between zero and fifteen units.

When the space bar is pressed down, the time is recorded. On each frame, the distance the bomb would be thrown is calculated. This is done by first taking the different in time modulo 3. This gives us a 3 second period. If the elapsed time is in the first 1.5 seconds, the distance is dt / 1.5s * 15. This results in some fraction of a distance starting at 0 and ending at 15 when dt reached 1.5s. If the elapsed time is in the second 1.5 seconds, the distance is 15 – (dt – 1.5s) / 1.5s * 15. This results in some fraction of a distance starting at 15 and ending at 0 when dt reaches 3s.

While the player is adjusting the distance, a transparent red circle is drawn on the ground plane in order to indicate the targeted distance. To draw this circle in its proper position, we take a unit length forward vector (of the camera/player) and multiply it by the target distance, as calculated above. We

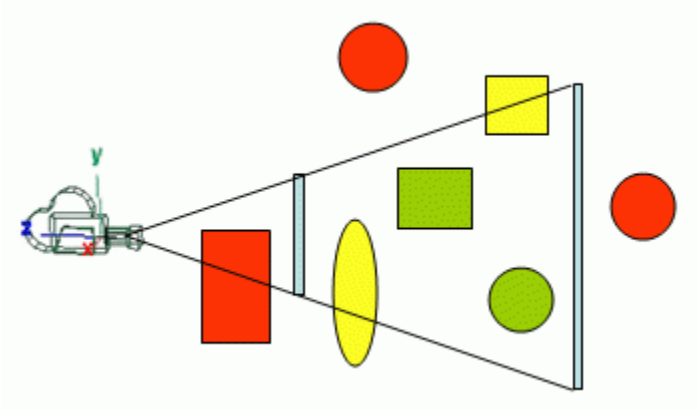
then add this vector to the characters position, which gives us the targeted position in world coordinates. This position is then the center of the circle which is drawn with the radius being the bombs explosion radius. This gives the player a clear indication of where the bomb will hit, and what will be in its explosion area.

When the player releases the space bar, a bomb object is created and begins its trajectory from the players position to its target position. When a bomb is created, it is given some starting position and some ending position. As an object, it also has a current position. On the bombs update function, it moves towards its destination position while testing for collision with objects. Once the bomb reaches its destination position, or it collides with an object, it explodes.

When a bomb explodes, it expands the radius of its bounding sphere to be the radius of the explosion, and then tests for collision against zombies. Any zombie that collides with this bounding sphere is then damaged.

5E. View Frustum Culling

The view frustum is everything that is potentially visible on the screen. This volume is determined by the settings of the camera, and since we are using a perspective projection it takes the shape of a truncated pyramid. One of the biggest slowdowns in the graphics pipeline is sending geometry to the GPU to be rendered. Why send geometry to the GPU when it is not going to be visible on the screen? That is the goal of view frustum culling – to only draw objects that are potentially visible.



The view frustum. Red objects are not visible and should be culled.

Green and yellow objects are potentially visible and can be sent to the GPU

The first step of view frustum culling is to extract the six planes that form the view frustum. To do this, we first grab the model and projection matrix from OpenGL. These two matrices are then composed into one combined matrix. We extract the planes in the form Ax + By + Cz + D = 0 and store the A, B, C, and D value of each plane.

Left Plane: A = m41 + m11, B = m42 + m12, C = m43 + m13, D = m44 + m14

Right Plane: A = m41 - m11, B = m42 - m12, C = m43 - m13, D = m44 – m14

Top Plane: A = m41 - m21, B = m42 - m22, C = m43 - m23, D = m44 – m24

Bottom Plane: A = m41 + m21, B = m42 + m22, C = m43 + m23, D = m44 + m24

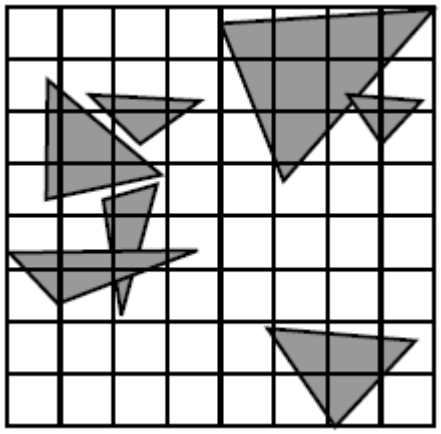
Near Plane: A = m41 + m31, B = m42 + m32, C = m43 + m33, D = m44 + m34

Far Plane: A = m41 - m31, B = m42 - m32, C = m43 - m33, D = m44 – m34

To then test whether a BoundingObject is inside the view frustum or not, we test the sidedness of each plane. To test a sphere, you make sure the distance from the plane to the center of the sphere plus the radius is negative for *all* planes. If it is positive for any plane, then the object is outside of the view frustum and does not need to be drawn.

5F. Spatial Data Structures

Collision tests are **everywhere** in the game. For every frame you move, you test for collision against every object in the world. Every time *anything* moves, it tests for collision against every object in the world. That is a whole lot of tests, and most of them can be avoided. Why test if you collide with a wall that is on the opposite side of the map? In order to prune out many useless collision tests, Zombs uses a type of spatial data structures called Uniform Spatial Subdivision.



Example of uniform spatial subdivision where each cell contains

a list of any triangle either partially or wholly contained by it

Every level consists of a 50x50 grid. When a map is loaded in, it is split into 100 5x5 “chunks”. Each chunk is a small region of the overall map and contains a list of all objects that are either partially or wholly contained inside that chunk. As objects move around the map, they are added/removed from the list of objects contained by the respective chunk(s). When testing for collision, it is now only necessary to test for collision against the list of objects in your chunk(s).

5G. Wall Transparency(Xray Weapon)

While transparency may seem like a simple concept in the real world, it was a rather difficult concept to implement because of the steps necessary to properly alpha blend and make objects

transparent. The first introduction of transparent objects was the (no longer existing) transparent walls when a wall was between the camera and the character. In order to make a wall transparent, it takes more than just setting the wall's alpha component to something less than 1.0. The drawing of all objects had to be split into two passes. On the first pass, all opaque objects are drawn. Every object has a *transparent* boolean to indicate if it is in a transparent state or not. After all opaque objects are drawn, the second pass sorts the remaining objects by distance from the camera. This is done using the built in sort() method, however I had to design a function used for the comparison.

sortObjectsByDistanceToCam(Object* left, Object* right)

This function is pretty straightforward. It uses BoundingObject::distanceFromPoint() where the point is the cameras position. The function returns the distance from the camera to the edge of the BoundingObject, which is a bit more complicated than just sorting by distance of the center point of the object. It then returns a bool whether distanceLeft < distanceRight.

Once the remaining objects are sorted by distance, they are drawn from furthest to closest. This results in transparent objects being properly blended into the world.

The xray weapon is the newest weapon addition to Zombs. When activated, it turns all walls transparent by setting their *transparent* boolean to true. The two pass drawing algorithm does the rest of the magic of making the walls transparent.

6. Results

Over the past two quarters, we have created a complete real-time 3D video game from the ground up. Since we were a smaller team (in comparison to the other teams working on similar projects) we do not have as many stunning visual effects, but we managed to create a one-level zombie

apocalypse with complex models, animations, particle systems.

We faced two major problems in the development process of Zombs. First, we had a team member who was not contributing. This put us at a huge setback for the first quarter because all of the components being worked on at the time were critical for the game to function, and this member was not completing her parts. The second problem was having a lack of direction. We never really sat down to decide *exactly* what we want our game to be. The proposal started more as a bomber-man style top-down game, but we ended up with a more first-person shooter style game. I think it would have helped the development if we had spent the first couple of weeks designing exactly what is going to be in the game: the weapons, the level layout, the enemies, etc. This would have probably produced much cleaner code in the end since we would have known exactly how to break down pieces to get to the end result.

Most player feedback received was that the controls were not as easy to figure out as they should be. In response, we changed some of the key binds and added control hints on the HUD at the beginning of the level. Players were impressed with the models and animations, and the smoothness of the game.

Bomb (bottle) being thrown infront of the player

Bounding Spheres and Bounding Boxes being displayed in debug mode

Flame thrower particle effect

Freeze particle effect

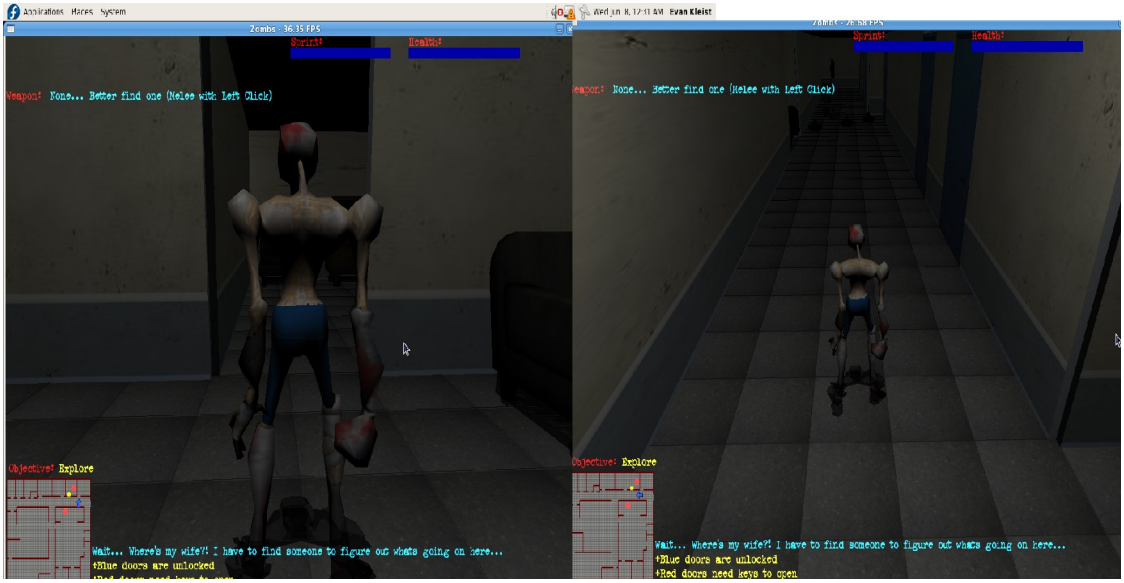
Melee combat

Various complex models

Sliding along a wall

Friendly interactive NPC's

Realistic shadows



Camera adjusting zoom to avoid obstruction

Xray weapon allowing for temporary vision through walls

Game over splash screen

7. Conclusion

The Zombs project was a great opportunity to put a lot of what I learned at Cal Poly to use. As a team, we designed, prototyped, implemented, and tested a complete video game. This was all done from the ground up. Zombs is also the largest software project I have worked on. I learned a lot about group programming projects.

This was the first time in a group setting where I ever had a problem with a group member. Since it was our senior project, I really would have expected everyone to give it their 150%, wanting a good senior project in the end. Our team frequently found ourselves scrambling at the last minute to get everything done. I think what we really could have used was a better team-manager, or more guidelines for the team-managers to go by. The milestones were typically two weeks apart, yet it seemed like I

was the only one committing code to the repository until a few days prior to the milestone.

Working on Zombs also made me feel like a strong programmer. Throughout the development process, I was constantly approached by team-members to help them debug their code or figure out tricky math problems for algorithms. The team considered me their strongest programmer. Its hard to see how your programming skills are until your actually writing code that interacts with other peoples code.

8. Future Work

While I am really happy with Zombs and having built a game from scratch, I do not plan to expand on it. When we first sat down and began coding, we had very little idea what designing a game would be like and did not even know what exactly *our game* was going to be like. This lead to lots of poor design decisions that made later design a nightmare. If I were to expand on Zombs, I would begin from scratch, plan out all the game details *before* implementation, and be able to produce much cleaner code now that I know what making a game really entails.

9. References

David, Henry. "C OpenGL The MD2 Model File Format." *Tfc.duke.free.fr*. Web. 08 June 2011. <http://tfc.duke.free.fr/old/models/md2.htm>.

Lighthouse3D. "Lighthouse 3D Tutorials." *Lighthouse3d.com*. Web. 08 June 2011. <http://www.lighthouse3d.com/>.

NeHe. "NeHe Tutorials." *NeHe Productions: Main Page*. Web. 08 June 2011. <http://nehe.gamedev.net/>.