ZOMBS*

3D Real-Time Game

Graphics Senior Project

Alan DeLonga
CSC 476++
Project Advisor: Zoë Wood
Spring 2011

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, surpassing both music and movies.

The present popularity and demand for video games, and the opportunity to put to practice many essential elements of Computer Science such as design, leadership, teamwork, and implementation is what led me to develop a game for my senior project. 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 worldwide.

The Beginning

The *Zombs* project was initially conceived as a proposal I gave, to our CPE476++ class, of a game that would be a mix between bomber-man and Diablo II, using cell shading. As the project progressed we chose to follow a more ominous ambiance with our models, lighting, camera angle and background sounds. I was chosen to be the leader since I came up with the game idea and proposal. By keeping a democratic atmosphere, and allowing all members to give input, our project became a cohesive culmination of our ideas. Our team was initially 5 but due to the inabilities of one of the members we only had 4 for the final quarter. Our members are Evan Kliest, Jordan Gasch, Reece Engle and me as project manager.

As we started the project we were given a list of technologies that were mandatory to incorporate into the game in particular:

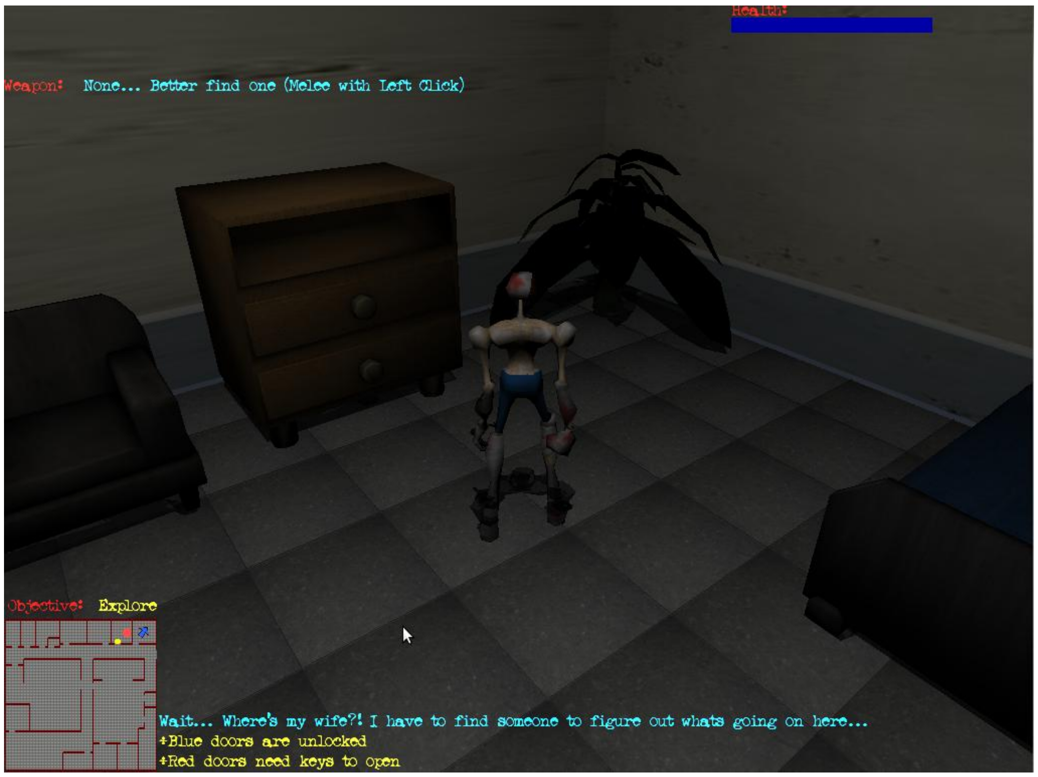
- Real-time movement/update (All)
- View frustum culling (Evan Kliest)
- Particle generation (Alan DeLonga)
- Spatial data structures (Evan Kliest)
- Per pixel shading (Jordan Gasch)
- Collision detection (Evan Kliest)
- Models *loading and creating* (Alan DeLonga)
- Animations (Alan DeLonga)
- AI (Jordan Gasch)
- Shadows (Jordan Gasch)
- Sounds (Alan DeLonga, Evan Kliest)

(to the right are additional technologies)

>Level Editor (Reece Engle)
>Smart Camera (Evan Kliest)
>Bomb Throwing (Evan Kiest, Alan DeLonga)
>HUD (Alan D., Reece E., Evan K.)
>Inventory (Alan DeLonga)
>Wall Transparency (Evan Kliest)

Along with these technologies we had to create a cohesive gaming experience. Later I will go over these different technologies, along with additional ones, and how they were incorporated into our game. Since each member attributed to different aspects I will only go into detail on the parts I helped integrate.

The game was coded using C++and used the libraries; SDL, libSDL_mixer (for sounds), libSDL_ttf (for text), libfreetype, libGLEW(for models, and shading). Our game was inspired by Diablo II's camera view and model interaction. We then altered it by locking the camera behind the player, dropping the angle, and pulling the camera in (instead of above) when the camera collides with walls.



None... Better find one (Melee with Left Click)

Wait... Where's my wife?! I have to find someone to figure out whats going on here...
+Blue doors are unlocked
+Red doors need keys to open

Game Play

The basic idea behind the story is you are a survivor in a zombie apocalypse. You wake up in a hospital after being in an accident, your main objective being to find your wife, who was in the car with you. As you progress through the level you realize something has gone horribly wrong, and that zombies have over run the hospital. Throughout the game you are updated with the inner thoughts of your character which help lay out this story line and give hints to game play aspects. The game is centered on following the objectives which aid you in finding weapons and items to fight off the zombie hordes. By using the weapons and items you navigate through the floors of the hospital to try to find your wife. The game uses "wasd" to move, mouse for camera rotation, left click melee and space bar weapon use.

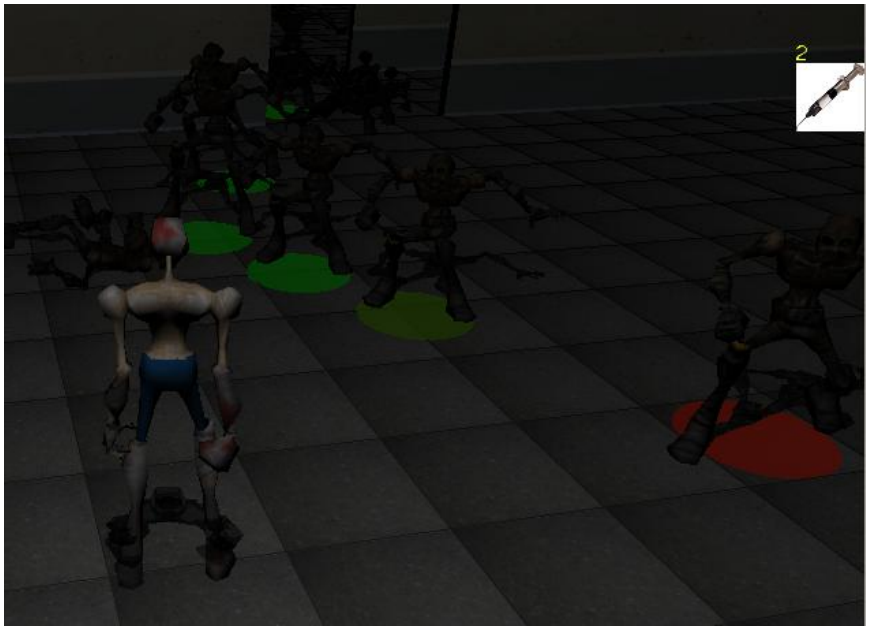
Incorporated Technologies and Technical Aspects

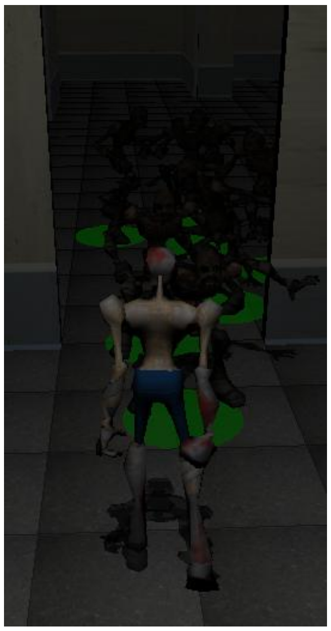
Level Editor

The first thing we saw an obvious need for was a level editor. This allowed us the ability to develop multiple levels populated with different items, objectives, and enemies. The editor sets up a basic grid of 50x50 and prints out a file containing all open areas for the enemy AI to work with. The editor also creates and loads files containing the locations of all the walls, items, and enemies. This allows for easy level development as well as the transitioning between levels during game play.

Artificial Intelligence

Our game uses a modified A* algorithm for the artificial intelligence of the zombies. The A* algorithm traverses the 50x50 grid which makes up the level and finds the lowest cost path to the player. This algorithm was simple, expandable, and allowed us the most accurate movement for enemies throughout our levels. Currently it is possible for our zombies to detect and find a path to the player from any position on the map with minimal slowdown to our game in its current state.



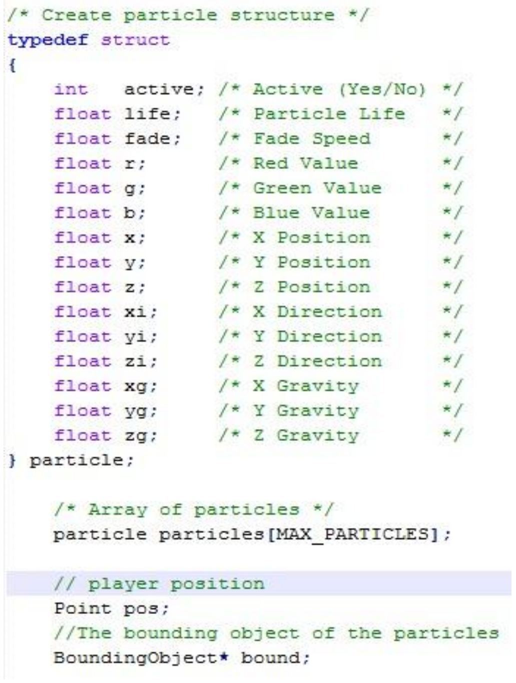


This algorithm is extremely effective because of the 2 dimensional plain our game plays in. This solution allows us to create attractors which have varying attraction levels. The attraction level determines how far away the zombies can detect the player from. Given more time our team planned to use this same algorithm for incorporating non playable character companions which would accompany the player through the level.

Particle Generation

Particle generation is a simple concept and adds a great deal to the aesthetics, but turned out to be more complicated in implementation and integration into our game. To start you have a generating point and particles. Each particle has a set of components (unique to each particle) including but not exclusive to its position, direction, velocity, color, life, and external forces such as gravity.(2) The particle generator sets up each particle with not only its start position but a random velocity and direction, each can be constricted within ranges to give different effects. While the particles are being update you apply each particles velocity, in its given direction, to update its position.(2) You also modify the other properties of the particle depending on what you decided to add to them. Ours only used life, direction and color. The life's decrease rate (fade) affects the distance of the particle stream from its generating point, once the particles life is 0 its position is reset to the generating point.(2)

Figure : Fire extinguisher



```
/* Create particle structure */
typedef struct
{
    int   active; /* Active (Yes/No) */
    float life;   /* Particle Life   */
    float fade;   /* Fade Speed      */
    float r;      /* Red Value       */
    float g;      /* Green Value     */
    float b;      /* Blue Value      */
    float x;      /* X Position      */
    float y;      /* Y Position      */
    float z;      /* Z Position      */
    float xi;     /* X Direction     */
    float yi;     /* Y Direction     */
    float zi;     /* Z Direction     */
    float xg;     /* X Gravity       */
    float yg;     /* Y Gravity       */
    float zg;     /* Z Gravity       */
} particle;

    /* Array of particles */
    particle particles[MAX_PARTICLES];

    // player position
    Point pos;
    //The bounding object of the particles
    BoundingObject* bound;
```

Figure : Particle generator elements

Figure : Flame thrower

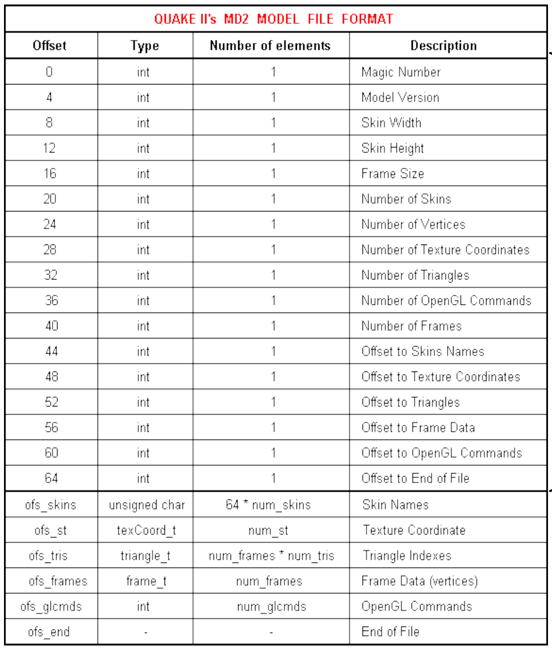
The biggest issue we ran into with the particle generator was getting the particles to blend and interact with the world properly. Our specific problem related to getting the alpha blending to happen correctly, which turned out to be an issue with ordering rendered objects in the scene. For alpha blending to work properly you draw all solid objects first, then whatever you want to be transparent over those objects last. In our specific case we had issues with getting the generator to be an aggregate object in the object list, for ordering. We then tried a generator which added each particle into the list of objects with the same result. We were able to manually order the generator in the world so that it would blend properly as a weapon. But we were unable to get one generating points in the world without it blending over all other objects.

MD2 Model Loading

All of the models in our game are in MD2 format. We chose MD2's because of the low vertex/polygon count and the availability of free MD2 models online. Having a low vertex/polygon count allowed us to populate the current view frustum with large amounts of moving models, as well as background objects, and shadows without bogging down the frame rate. We found tutorials and source code related to MD2 model and texture loading . We used these tutorials to integrate this technology into our game, which allowed us to load in various models and apply different skins to each. The major types of model groups in the game include background furniture, dead bodies, enemies/zombies, items, interactive non playing characters, and the player.

I used "The Quake II's MD2 file format" written by Devid Henry, dec 2002. This tutorial lays out all the components necessary for loading in MD2 models, skins/textures, and animation frames. I will just briefly overview how an MD2 file is formatted and how we get a model from that set of data. A deeper explanation into rendering, as well as texture mapping and animation interpolation, can be read in more detail in the above reverenced tutorial.

First off we must understand how the MD2 file is set up so we can then see what kinds of structures we will use to contain those sets of data. Once the file set up is known it is just a matter of parsing the file.



QUAKE II's MD2 MODEL FILE FORMAT

Offset	Type	Number of elements	Description
0	int	1	Magic Number
4	int	1	Model Version
8	int	1	Skin Width
12	int	1	Skin Height
16	int	1	Frame Size
20	int	1	Number of Skins
24	int	1	Number of Vertices
28	int	1	Number of Texture Coordinates
32	int	1	Number of Triangles
36	int	1	Number of OpenGL Commands
40	int	1	Number of Frames
44	int	1	Offset to Skins Names
48	int	1	Offset to Texture Coordinates
52	int	1	Offset to Triangles
56	int	1	Offset to Frame Data
60	int	1	Offset to OpenGL Commands
64	int	1	Offset to End of File
ofs_skins	unsigned char	64 * num_skins	Skin Names
ofs_st	texCoord_t	num_st	Texture Coordinate
ofs_tris	triangle_t	num_frames * num_tris	Triangle Indexes
ofs_frames	frame_t	num_frames	Frame Data (vertices)
ofs_glcmds	int	num_glcmds	OpenGL Commands
ofs_end	-	-	End of File

```
// md2 header
typedef struct
{
    int     ident;      // magic number. must be equal to "IDP2"
    int     version;    // md2 version. must be equal to 8

    int     skinwidth;  // width of the texture
    int     skinheight; // height of the texture
    int     framesize;  // size of one frame in bytes

    int     num_skins;  // number of textures
    int     num_xyz;    // number of vertices
    int     num_st;     // number of texture coordinates
    int     num_tris;   // number of triangles
    int     num_glcmds; // number of opengl commands
    int     num_frames; // total number of frames

    int     ofs_skins; // offset to skin names (64 bytes each)
    int     ofs_st;    // offset to s-t texture coordinates
    int     ofs_tris;   // offset to triangles
    int     ofs_frames; // offset to frame data
    int     ofs_glcmds; // offset to opengl commands
    int     ofs_end;    // offset to end of file

} md2_t;
```

From this table we can see there are 3 basic sets of data; an offset to a group of data, the amount of data/elements contained in specific structure, and the structures themselves. Notice that the structures all have variable offsets to allow for any different sizes of data set that can be represented by a model. This is what gives the great flexibility in the diversity of models that can be saved in this format.

First we have to create classes to represent each set of data we are getting from the file. We can do this by grouping all the offsets and element numbering into a single header file structure:

When reading in this data it should be noted that, the first component is the "magic number." This value is used to check if it is a valid MD2 file. If this variable is not equal to "IPD2" then it is not an MD2 file, so you can close the file. The next variable indicates what file version it is, which must be 8. The rest of the values are the amount of elements in each structure and offsets to the beginning of each structure set. What we have left in the file are the structures correlating to most of the offset names in particular; `ofs_skins` points on model's texture names, `ofs_st` on texture coordinates, `ofs_tris` points on vertices, `ofs_frames` on the first frame of the model. (1)

The main structures we need are as follows. First we will need a data type for vector, which is standard implementation for most 3D programs. For this instance our vector only has to contain three floats, for x, y, z components. Our actual implementation of vector was much more robust, including common operations on vertices.

```
typedef float vec3_t[3];
```

Each model has a vertex count equal to the amount of frames multiplied by the number of vertices, (num_frames*num_xyz). Each of these vertices are stored with a position and light normal vector for lighting.

```
// vertex
typedef struct
{
   // compressed vertex (x, y, z) coordinates
   unsigned char   v[3];

   // index to a normal vector for the lighting
   unsigned char   lightnormalindex;

} vertex_t;
```

The next two structures are for textures and animation which I won't go too far into, but are included for completeness.

```
// texture coordinates
typedef struct
{
    short    s;
    short    t;
} texCoord_t;
```

’s’ and ‘t’ are divided by the header’s skinwidth and skinheight to generate the proper float mapping of the texture coordinates.

```
// frame
typedef struct
{
    float       scale[3];       // scale values
    float       translate[3];   // translation vector
    char        name[16];       // frame name
    vertex_t    verts[1];       // first vertex of this frame

} frame_t;
```

Frame stores the information for setting up the orientation of each vertex per frame. ‘verts[1]’ refers to an array of vertices, where verts[num_xyz -1] would be the last vertex. So for updating each vertex from this structure we would have the following formula:

```
vertex.x = (frame.verts[i].v[0] * frame.scale[0]) + frame.translate[0]
```

Which is done for each x, y, z component correlating to its 0, 1, 2 index into the frame’s member arrays, indexing ‘i’ from 0 to (num_xzy-1).

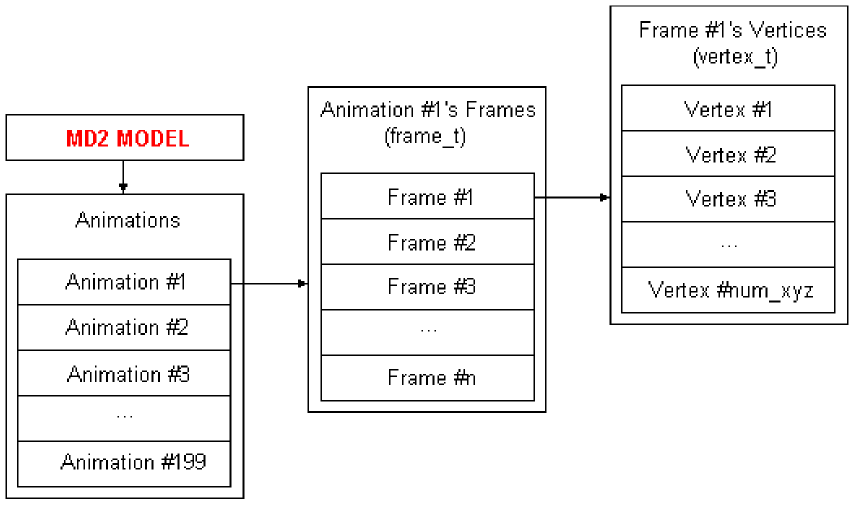


Figure : Showing the connection between animations, frame sets, and vertex sets (1)

Next we set up the structure that will link the texture coordinates to the vertices. These are paired in triplets, to create a triangle mesh.

```
// triangle
typedef struct
{
    short   index_xyz[3];    // indexes to triangle's vertices
    short   index_st[3];     // indexes to vertices' texture coorinates

} triangle_t;
```

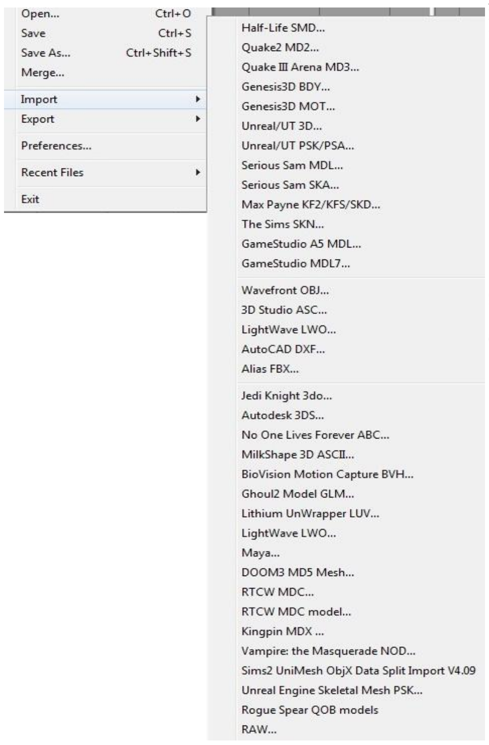
By making Verticies[ ] as an array of vertex_t, TextCoord[ ] as an array of textCoord_t, Meshes [ ] as an array of triangle_t and anorms [ ] as an array of vec3_t (which stores all the precalculated normal vectors). We could use the below method to draw the model. This method can be abstract at first glance, and uses GL_TRIANGLES, to get better performance with GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN these can implement using the OpenGL commands.

```
glBegin( GL_TRIANGLES );
  // draw each triangle
  for( int i = 0; i < header.num_tris; i++ )
  {
     // draw triangle #i
     for( int j = 0; j < 3; j++ )
     {
         // k is the frame to draw
         // i is the current triangle of the frame
         // j is the current vertex of the triangle


          glTexCoord2f(
         (float)TexCoord[ Meshes[i].index_st[j] ].s / header.skinwidth,
         (float)TexCoord[ Meshes[i].index_st[j] ].t / header.skinheight );

          glNormal3fv(anorms[ Vertices[ Meshes[i].index_xyz[j] ].lightnormalindex ] );

          glVertex3f((Vertices[Meshes[i].index_xyz[j] ].v[0] * frame[k].scale[0]) +
                        frame[k].translate[0],
                       (Vertices[ Meshes[i].index_xyz[j] ].v[1] * frame[k].scale[1]) +
                      frame[k].translate[1],
                     (Vertices[ Meshes[i].index_xyz[j] ].v[2] * frame[k].scale[2]) +
                      frame[k].translate[2] );
      }
  }
glEnd();
```

That concludes the data structures necessary for file parsing an MD2 model. From here a class is constructed to represent the model and functions are made for reading, storing, drawing (using OpenGL Commands), animating, and texturing. For more information and greater detail on these last steps visit the tutorial state above, at http://tfc.duke.free.fr/old/models/md2.htm. All information and ideas pertaining to model loading, expressed in this paper, have come from this site.

MD2 Model Creation/Animation/Skinning

Along with integrating the model loader I also worked on the construction, skinning and animation of the models through MilkShape. MilkShape is a modeling program like Maya or Blender. The reason for choosing MilkShape, as opposed to Maya or Blender, was because of its import/export functionality. MilkShape is capable of importing and exporting to over 45 different file formats including but not exclusive to: MD2, MD3, MD5, OBJ, ASCII, 3DS, Maya, RAW, TEXT, as well as formats from playstation, warcraft III, unreal tournament, and the sims.



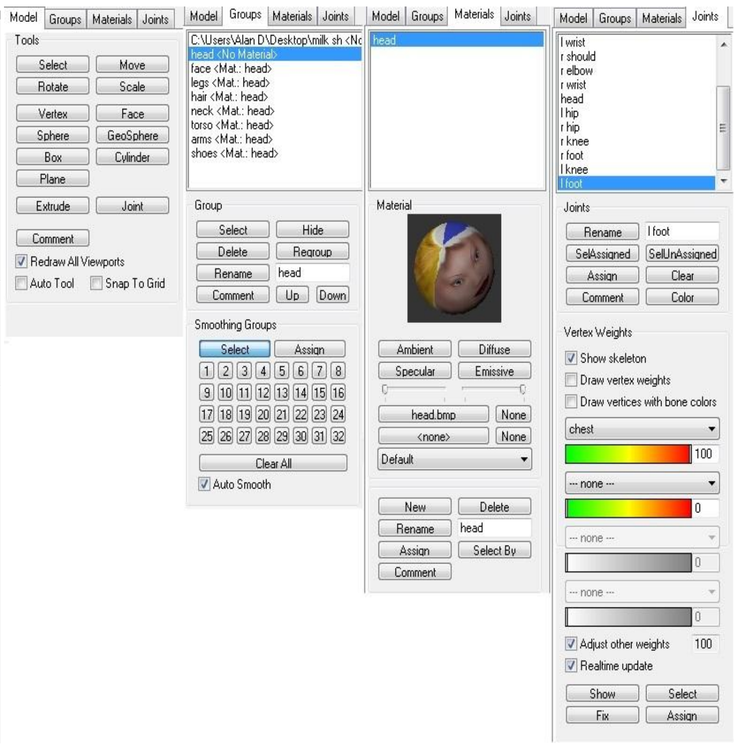


Figure : MilkShape Import format selecitions

Figure :MilkShape tabs for model creation, skinning and animation

MilkShape also has a very simplistic interface. This interface breaks up the four main aspects of modeling: creating a model from vertex/face creation and primitive shapes; setting up different grouping of vertices (labeling groups such as head, torso, arm, etc.); loading in and assigning different textures to vertex groups labeled previously; lastly making joint skeletons and associating them with vertex groups to get the correct movement of the models mesh with that of the skeleton, for animation.

For animation it allows you to set key frames for the placement of the model and then deals with the interpolation internally to output the necessary points for each vertex, for each animation frame. By setting up joint skeletons you are able to attach vertices to joints for movement. Each joint is connected hierarchically so rotations and translations affect all joints that are with less precedence from your root joint. Each animation is made by translating and/or rotating each joint to get the desired movement, then saving the frame. The program deals with the interpolation between key frames to make movements fluid through frames.

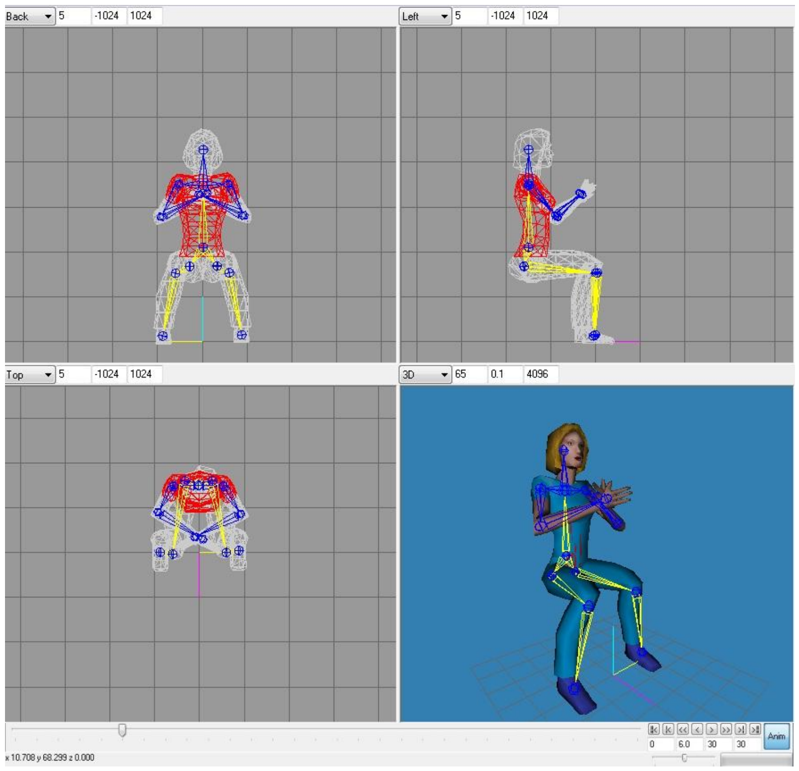


Figure: In the picture above the torso (vertex group) is selected which has the blue “shirt color”, textured to it.

Similarly, the skins have to be attached to a set of vertices for them to be correctly oriented on the model from its assigned material (image file).

```
anim_t CMD2Model::animlist[ 21 ] =
{
    // first, last, fps
    {   0,  39,  9 },   // STAND
    {  40,  45, 10 },   // RUN
    {  46,  53, 10 },   // ATTACK
    {  54,  57,  7 },   // PAIN_A
    {  58,  61,  7 },   // PAIN_B
    {  62,  65,  7 },   // PAIN_C
    {  66,  71,  7 },   // JUMP
    {  72,  83,  7 },   // FLIP
    {  84,  94,  7 },   // SALUTE
    {  95, 111, 10 },   // FALLBACK
    { 112, 122,  7 },   // WAVE
    { 123, 134,  6 },   // POINT
    { 135, 153, 10 },   // CROUCH_STAND
    { 154, 159,  7 },   // CROUCH_WALK
    { 160, 168, 10 },   // CROUCH_ATTACK
    { 196, 172,  7 },   // CROUCH_PAIN
    { 173, 177,  5 },   // CROUCH_DEATH
    { 178, 183,  7 },   // DEATH_FALLBACK
    { 184, 189,  7 },   // DEATH_FALLFORWARD
    { 190, 197,  7 },   // DEATH_FALLBACKSLOW
    { 198, 198,  5 },   // BOOM
};
```

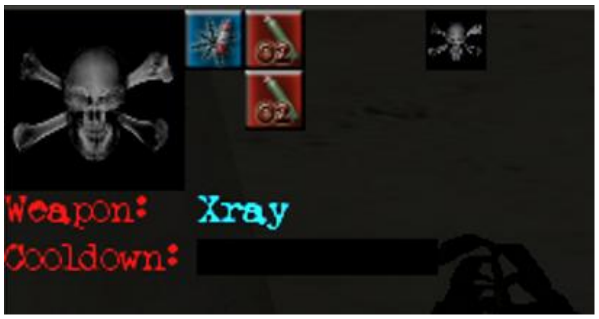
It was only after working out animations for a few models, and attempting to integrate them into the game, that I ran into the limitations of MD2 animations. MD2 models have locked keyframe sets. For example each animation is a frame range in the animation array, as shown in the code to the left.

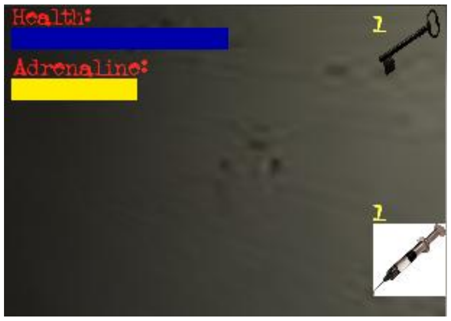
One problem that arose from this set up was our inability to modify the ranges set by the MD2 format, in MilkShape. Since an MD2 animation set is under 200 frames long and contains 21 separate animation sets it only gives you between 6-14 frames for each animation. For the animations to look fluid interpolating between each movement I was using between 45-120 frames for each animation. We were unable to fully find a fix to this issue but we found that if we only set one animation, it could take up the entire frame set. But we were unable to set up a second segment of animations after the first. Specifically when indexing into the animation array it was able to access 0, but it would segfault if you tried to use any other number to index into the array.

The second issue with MD2 model's animation is it does not blend between animations. For example MD5 models have 3 separate segments for animation; the head, torso, and legs. This allows you to set separate animations for each without the others being affected. This allows you to have the model running and shooting, then hitting while still running, then just running without having to create completely separate animations sets for each full body animation. To make it clearer, each separate segment has its own animation array of keyframe sets. To achieve this with MD2 you have to create the legs, torso, and head as completely separate models, set up the animation keyframe sets for each, load them in separately and combine them as a complete model in your application/game. Because of the issues described above, with not being able to create keyframe set ranges, we were unable to incorporate this fix in our game.

Inventory

Our game also includes an inventory that automatically combines, creates, and makes weapons available for use. The system is set up to make a weapon available for selection as soon as all necessary components have been collected. Once the weapons are available they are set to appear in the upper left hand corner. The currently selected weapon is shown in the left corner; all available weapons are show in their upgrading tiers to the right of the current weapon. Below the currently selected weapon its name and associated cooldown are displayed. Each weapon is set up with different use and cooldown weights to reflect the power of the item, and balance them with game play.

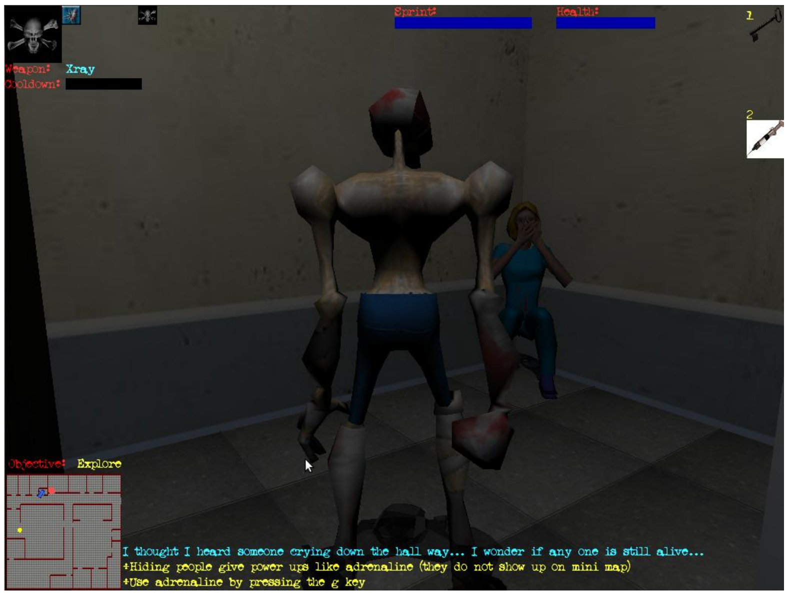




Other usable items, like keys and adrenaline, are displayed in the opposite corner. The game was initially conceived as an item collection game, where you would be able to pick up a plethora of items to combine into a wide range of weapons. The complexity of developing user knowledge of available weapons and what items it took to combine led us away from this idea to one that was automated.

Heads-Up Display

To give the player more knowledge of all the aspects of the game we chose to add a multi faceted HUD. Along with the weapons and inventory, the heads up display (HUD) also contains the mini map which shows your current position and orientation, the current objective location, and any enemies within a radius of the player. Above the mini map we have the current objective. To the right of the mini map are inner thoughts/story line in blue, and hints/game-play help appear in yellow. In the bottom right corner we have a zombie awareness indicator. Currently it is a zombie picture that changes through 6 color sets, portraying the alertness of zombies in the level. The higher the awareness the faster the zombies move, and the longer the allowed path distance for the zombie AI to get to the player becomes. This means at the highest awareness level zombies move faster than the player, unless you are sprinting, and all the zombies in the level will be aware of the player and be trying to get to them. In the upper right the collected non-weapons are shown. Currently this contains 3 different types of keys and adrenaline injection. Lastly just to the left of the keys is the player's health and the adrenaline timer bar shows up under the health, when activated.



I thought I heard someone crying down the hall way... I wonder if any one is still alive...
+Hiding people give power ups like adrenaline (they do not show up on mini map)
+Use adrenaline by pressing the g key

As well as restructuring the HUD, I also helped create the structure for the objectives. Setting up where they are located, and the inner thoughts and hints displayed after each objective is hit. When followed, the objectives set up a story that leads the player through the level and into the various interactive objects throughout the level. All necessary controls are displayed through the yellow hints given to the player after each objective is hit.

One of the biggest game play problems we came upon was when players wanted to play without paying attention to the hints. This made them unaware of how to use most things and hindered their ability to understand what was going on. In retrospect having a button layout on the screen and narration instead of the text would have been better to inform the player when objectives were hit, which would allow them to pay more attention to playing instead of trying to read.

Collision Detection

We were presented with a complicated problem in representing the interactions between the variety of items, weapons, and moving models we chose to add. Our 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, 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. This was primarily used in detecting whether the camera's view of the player was obstructed.

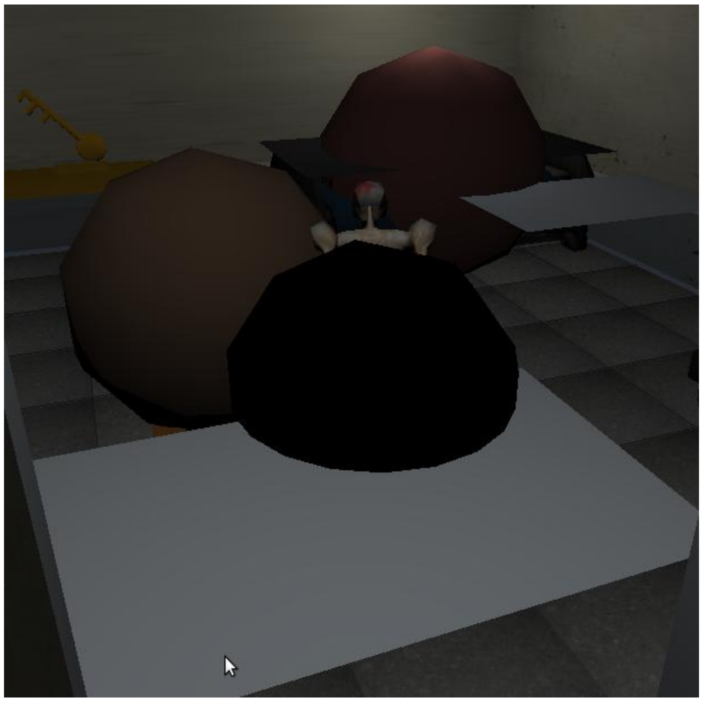
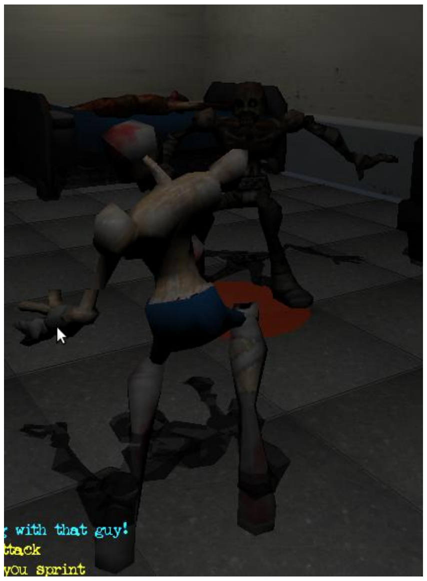
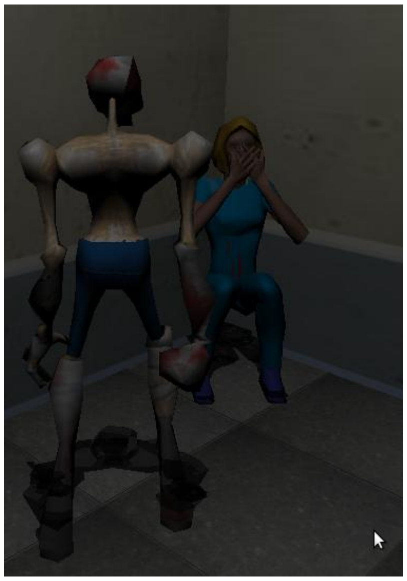


Figure : shows collision debug mode (in which representations of all bounding objects are drawn)

Most of my contributions to the collision detection were in setting up the reactions with the collisions between the objects, and enemies, in the game. One aspect was restricting the damaging angle of the flamethrower. This was done by using a bounding sphere to surround the flame object, particle generator. Once a collision was made with this bounding sphere the center point of the collided object is taken. From there an angle is calculated between the forward vector of the player and the point of collision. If the point being collided with was within the angle range of the flame thrower (20 to -20 degrees to either side of the forward vector) it would damage the object, if it could take damage. Another fix that was added was creating a separate set of bounding spheres and checks for enemies, for testing collision between each other, to allow them to be closer to fit through doors. As part of the reactions to collision I set up all the animations that went along with collision; such as the zombie pain reaction, crawlers awakening, and non playable characters interactions. The rest of my work was in setting up all the sound cues for the different types of interactions in the game between the player and enemies, special items, and background objects.

Shadows

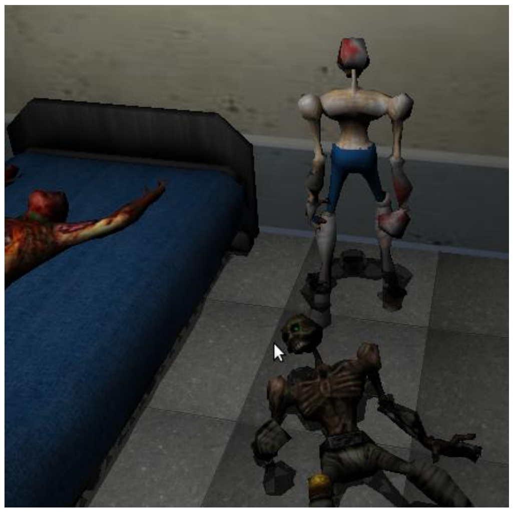


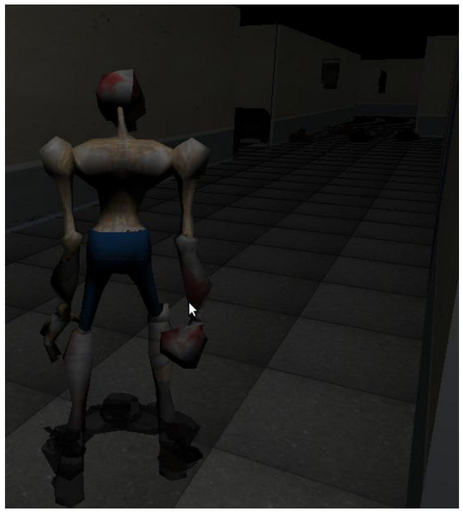


To help attribute to our realistic feel we first explored adding shadow volumes. When this was found to be too large of a project for the time we had we looked into projected shadows. We ended up implementing a shadow engine that uses projected shadow mapping to represent shadows. To add to the horror ambiance and limited field of vision we decided to lock a dim light behind the player. This helps highlight the shading on the character and gives the shadow a stationary light source to use in rendering. Shadows greatly added to the aesthetics and overall feel.

Per Pixel Lighting

Lighting is a very important component attributing to the atmosphere, game style, and ambiance of the game. The Opengl fixed function graphics pipeline did not allow us the flexibility we wanted for manipulating lighting, so we used Opengl's shader language GLSL to implement a per-pixel lighting system. Our original plan was cell shading, which is the process of clamping the lighting values so that once a thresh hold is crossed the lighting value changes drastically, giving our game a comic book feel. However, we found that the ambiance of our game did not fit with this style so we implemented a more realistic lighting model.





The method we used for lighting is Phong shading which computes the ambient, diffuse and specular lighting combines these values and multiplies it by the color value stored by Opengl. This allows our game to be very dark and have realistic per pixel lighting, rather than per vertex lighting. Lighting attenuation was another integral component for the look and feel of our game. This gave our lights the ability to grow dimmer from farther away, rather than having a constant ambient level for every light at every position.

Sounds

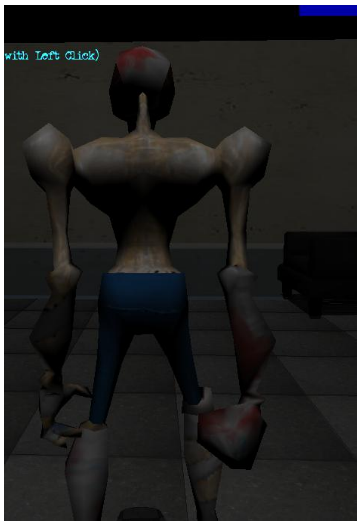
Having sounds that draw the player into the moment is an important aspect of our game. We did this by featuring over 30 different sounds, including 7 sounds looping for background ambiance. There are sound cues for various actions and situations including but not exclusive to; Dead bodies, doors, crawlers, zombie pain and groan, non-playable character hiding and found reactions, player pain, melee, heartbeat, use of adrenaline, and sprint.

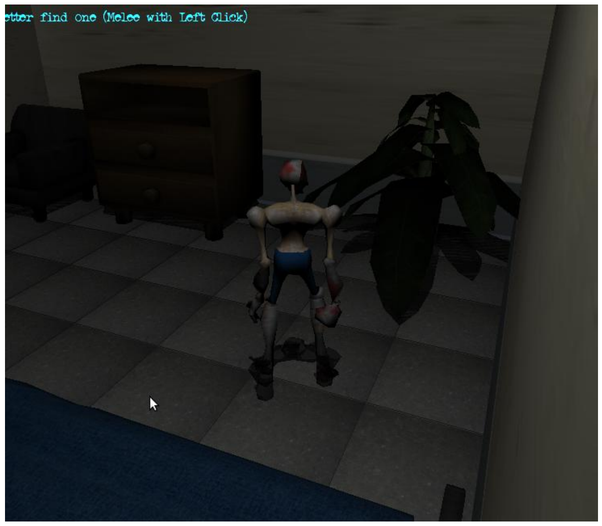
We used the SDL library for its api on sound and text display management. We did run into issues with the limitation of the number of available channels. Since we have so many sounds going on at any given time some get kicked out of their channel before finishing. We tried to solve this issue by allocating more channels, but we seemed to be constricted to 8 running channels once.

View Frustum Culling

The view frustum culling algorithm helps increase our frame rate by eliminating the rendering of objects not in the camera view. The algorithm we used in CSC476 only dealt with bounding spheres, but we had to deal with bounding boxes as well. Our algorithm first computes the 6 planes of the view frustum, and then checks the sidedness of all objects in the world, if the object is inside the view frustum its sidedness is positive. Only if the object is inside of the view frustum is it drawn. This is a huge performance enhancer as only geometry that is potentially visible by the player is sent to the GPU.

Smart Zoom Camera





The camera in Zombs is a 3rd person aerial camera. For the first half, the camera sat high in the sky looking down at the player at about a 45 degree angle. This gave the player a “godlike” position where they could see over the tops of walls and into adjacent rooms. In the second half, we dropped the camera down lower from the sky so it was no longer able to see over the tops of walls. This helped put the player in the characters position and give the player a more desperate feel.

However, change now brought about the problem of the view of the player being obstructed by walls and other objects. In order to prevent this, the distance from the camera to the player would increase/decrease in order to keep the player always in sight of the camera. This was achieved by a line-bounding volume intersection test where the line was the line from the camera to the player. If it intersected any objects in the world, we then knew that the view was obstructed and the distance must be decreased.

Bombs (throwing and aiming system)

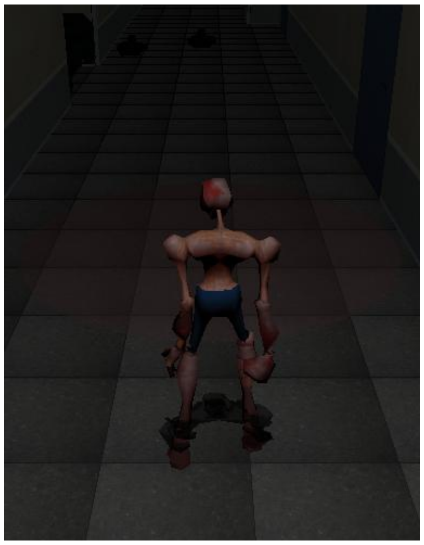


Figure : Aiming circle

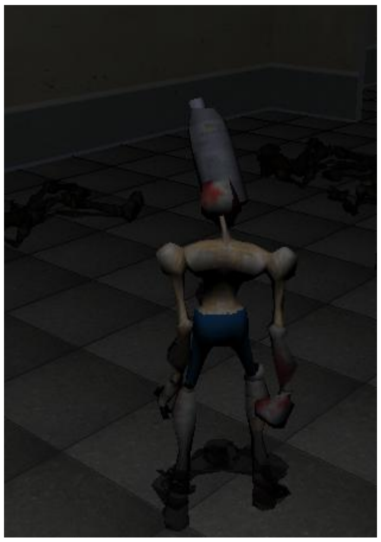


Figure : Bomb throwing

The only projectile weapon existing in the game is the ability to throw bombs. The bomb thrown straight in front of the player, and the distance the bomb is thrown is calculated by the amount of time the player holds the space bar. The distance oscillates from the players position to the position 15 units in front of the player, and this is visible in game by a transparent red circle moving forwards and backwards on the ground plane. Once the bomb is thrown, it is then an interactive object in the world. If the bomb collides with any object, it will then explode. If it never collides with an object, it will explode when it hits the ground plane at the distance aimed at.

Spatial Data Structures

Another form of optimization was the use of a spatial data structure. It is not necessary to always test for collision (and other things) against *every* object in the world, because the objects may be nowhere near each other. The spatial data structure implemented in our game was Uniform Spatial Subdivision. Upon initialization, the map (a 50x50 grid) is broken into several “chunks.” Each chunk is a

5x5 square of the map and contains a list of all objects contained within that chunk. As objects are loaded into and move about the map, they are added/removed to/from the list of objects in the respective chunk. If the object overlaps multiple chunks, it is added to the list of each of those chunks. This is a big optimization because now, for example, instead of the player testing for collision against every object in the map every frame, he only tests for collision against objects in the specific chunk he is in.

Wall Transparency (Xray)

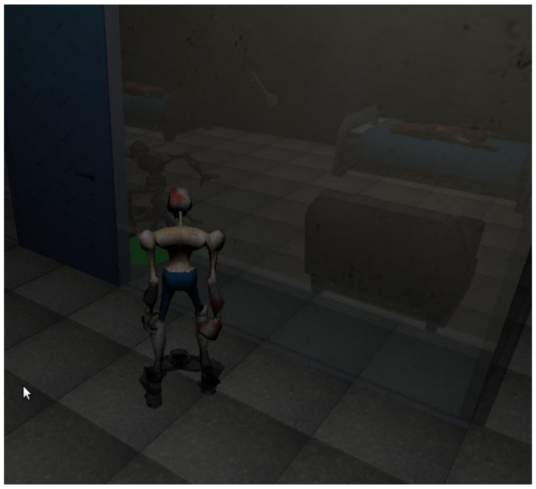


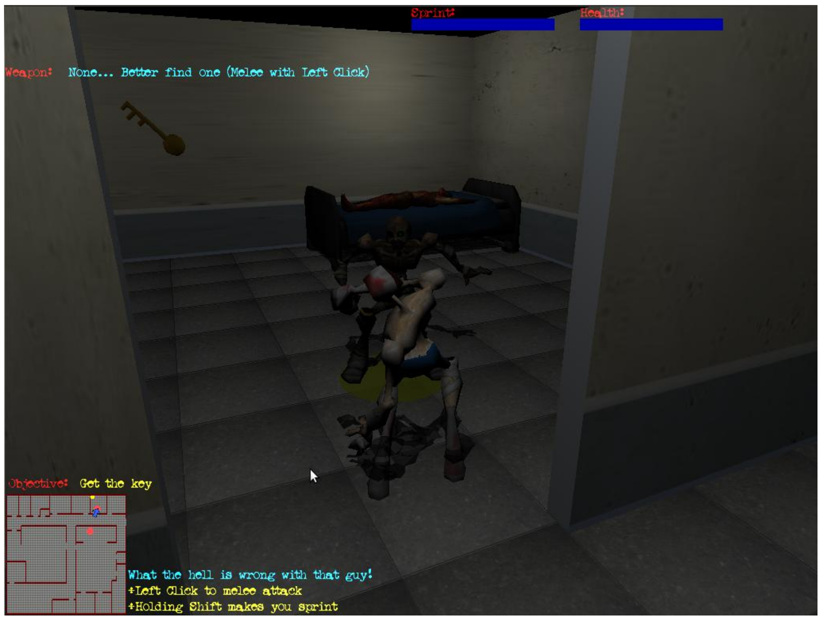
Figure : Using Xray (triggers walls to go transparent)

While transparency may seem like a simple concept, it was rather difficult to get transparent objects in our game because of the steps needed to properly alpha blend. It now became necessary to split the drawing of objects into 2 passes. On the first pass, all opaque objects are drawn. On the second pass, the remaining objects are sorted by distance from the camera, and then drawn from furthest to closest.

Results

At the end of 20 weeks we are proud to have come out with a cohesive playable 3D real-time rendering graphics game. We were able to persevere through hardships, spring back from dead ends in implantation, and navigate through hidden issues in integrating all of the technologies. We are very proud of the outcome of the overall feeling of the game, the depth, and scale we were able to achieve in the 20 week time span.

I am personally proud of all the technologies I was able to learn and integrate into our game under frequent deadlines. Our development process was fast paced and leaved little room for error. Even though we ran into many problems, we were always able to help each other figure out issues. Being a manager of a programming group is no small task, and I am glad that I was able to help the rest of the group achieve what we have done.



Weapon: None... Better find one (Melee with Left Click)

Objective: Get the key

What the hell is wrong with that guy!
+Left Click to melee attack
+Holding Shift makes you sprint

Most the issues we ran into as a group was due to a lack of time, because of the heavy course loads, and inexperience in implementing some of the technologies. One issue that was dealt with involved a team member that was unable to complete anything that was assigned. To help accommodate their lack of programming ability I assigned only technologies that were lab assignments for the class, to be integrated into our game. After a repeated inability to finish assignments, and attributing to repo issues, we gave them an ultimatum after the first quarter. During spring break they were allowed to choose something to implement to show their diligence and dedication to the project. At the end of the week the selected changes were only minor modifications, which again were not fully functional. Because of the lack of competence we were forced to let them go, and push through the second quarter as a 4 person group.

This experience helped bring us together and feel more accountable to each other. Even though we hit hardships, with putting in all the time necessary without totally neglecting our other classes, we were able to pull together and finish everything expected of us. There has not been one aspect that our group has decided to implement in our game and has been completely unable to produce. Some technologies were not as polished as we may have wanted, but all have been incorporated in a cohesive manner that really helped accentuate the game.

Conclusion

This experience has taught me a lot about myself and the discipline, communication skills, knowledge, and structure necessary to be a productive leader and group member. Managing a group of peers has presented many issues. The biggest being the inability to motivate some members when other things take priority. These difficulties mostly come from the rigorous work load of the computer science courses at Cal Poly. But learning to work in a stressful environment under strict deadlines builds the skills necessary to thrive in the work force.

It has also shown me that through hard work and perseverance anything is possible. When we started no one had a real idea of what we were about to embark on. With only a set of necessary technologies, we collaboratively decided every aspects of our game and conformed the content to adhere to these ideals. When managing I felt that giving everyone the power to give input on how they think the game should look, play, and feel would make the game more robust. All things were decided on as a group, and then individually implemented revealing our own personal touches. Giving everyone the ability to contribute to design decisions gives members a greater connection to, and pride in, the final product.

Most importantly this project has proved that it I am able to learn and produce products quickly in the midst of a hectic schedule. One of our greatest assets was the variety of experiences of our group members that allowed us to excel. Each person found their special niche to which this game would have been unachievable without. Evan was a great asset in how he was able to create the underlying structure for the files and connections. He was also our go too guy for specific compilation errors and implementing new technologies such as the transparent walls. Reece took on the daunting task of creating, and updating, our level editor and linking it with everyone else additions as they came. Jordan had taken classes in AI, and was concurrently taking a ray tracing course which helped him do all of our shading and shadows.

Because I felt it was my duty to keep everyone as productive as possible, I assigned myself the larger tasks that were not gone over in class. It wasn't until the second quarter, after they were already implemented, that models, animations, and particle generation were lectured on. With my artistic background I was able to pick up on creating, skinning and animating models. I implemented the model loader, as well as creating over 10 models, and animating 3 models. I had my hands in everything, besides the shaders, on this project and from that I have respect for what everyone has been able to produce. I also set up all the sounds, reactions, animations, weapons, power ups, and story line. At times I was unsure if I was going to be able to finish certain tasks, but in the end it all equates to the time you put in. Only fear, impatiens, and indecision can hold you back from reaching your goals.

Future work

I hope to continue development on this project to learn more about the aspects I wasn't as heavily involved in implementing. Having more time to spend on models and animations will allow me to create more enemies, items and non playable characters throughout the levels. I would also like to get particle generation blending to work for the bomb weapon, in particular getting multiple particle generators working within the scene. Having the game at the point it is now opens it to endless possibilities for further work. Especially the amount of weapons, sounds, models, and game play aspects that can be incorporated with the structure that is currently in place. Most importantly we need to generate an executable that can be played on different configurations. The reoccurring theme of the project being, there is never enough time to put in everything you want.

References

Our game has many influences from the games we have played and seen over the years. Although the initial concept was closer to Blizzard's Diablo II, it turned out with more of a Resident Evil, by Capcom, type feel. We initially had the camera higher and completely locked and allowed walls to go transparent when the camera angle was blocked, an idea straight from Diablo. We later found that by dropping down the camera we were able to add to the feeling of desperation to the game play. This helped reinforce the ambiance we were trying to represent.

Figure : Resident Evil 3 by Capcom

Figure : Diablo II by Blizzard

Many aspects of our game are pretty standard for its type such as our mini map, melee/weapon attack, item collection, following objectives to proceed through the level, and level changing. It is similar to many 3rd person type games pulling ideas, like zombie crawlers, from Call of Duty's zombie levels.

Bibliography

1) The Quake II's MD2 file format" written by Devid Henry, Dec 2002http://tfc.duke.free.fr/old/models/md2.htm

2) Lesson 19: Particle generator tutorial, 1997-2006 Gamedev.net http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=19