ABYSSAL MARAUDERS FINAL REPORT

Jia-Jie (Joel) Chen

Victor Naso

Chuan Chang

Clive Lin

Timotius Margo

Electrical Engineering 478

The University of British Columbia

April 14, 2008

Table of Contents

List of Figures

1.0 Game Overview

1.1 Game Concept

Abyssal Marauders was originally designed to be a 3D version of *Space Invaders*, hence the synonymous game name. Since its conception, Abyssal Marauders has evolved into a 3D space action game where the player assumes the role of a pilot who must destroy incoming enemy kamikaze ships before they explode. Each time an enemy collides with the player ship, the player loses hit points. Once the player loses all of their hit points, the game ends.

The purpose of the game is to out-manoeuvre the enemy AI and annihilate a maximal number of enemy ships before being destroyed. The player receives points based on the type of ship destroyed as well as firing accuracy.

1.2 Game Engine Principles

1.2.1 General Game Engine Principles

The high-level conceptual overview of our game engine is as follows:

```
While the user is playing and does not wish to exit
{
    Check for input from the user
            Deal with any input quickly
    Check to see if it is time for a periodic update
            If it is time, deal with the periodic update
            If it is not time, render a frame to the scene
}
```

The reasoning behind this game engine stems from the recognition of three important factors:

1. User input cannot be missed and must often be acted upon immediately.

2. There are numerous game state alterations that do not require immediate re-evaluation partly because the game state is not static but dynamic, as discussed in subsequent sections.

3. The frame rendering rate has the lowest priority, but achieving the highest possible frame render rate translates into smoother animation and is desirable. Thus, rendering should occur at any opportunity, and therefore should scale with the processing power of the host machine.

Note: A common term used throughout the game engine section is “game state”. The precise definition of “game state” will be explained in future sections. Until defined explicitly, “game state” refers to a series of variables that determine the game’s current status (e.g. the position and heading of ships, weapons that are firing, ships that are hit, etc…). As game events transpire, game state variables change (e.g. the player changes heading, fires a weapon or a ship explodes) thus altering the game state.

1.2.2 Event Driven Vs. Periodic Game Updates

Clearly because Abyssal Marauders was a real-time action game our team found “pausing” the game difficult (i.e. waiting for input from the player while not processing game state variables with changes in time) unlike a board game or a turn-based strategy game. The primary issue is

that the game must constantly be in motion regardless of input, or lack thereof, from the user with the possible exception of a deliberate pause function.

Thus, our team opted for an event driven system where a polling loop constantly checks for new queued events and immediately responds to events in FIFO order. User input immediately generates new events. If there are no new events on the queue then the game state still progresses.

However, an event-driven methodology results in possibly conflicting requirements such as reacting quickly to user input while minimizing response periods. To avoid some of these conflicting requirements our team recognized the paradigm of capturing user input in real-time but not necessarily evaluating the input's consequences in real-time. Therefore, the user input capture rate can be significantly greater than the input response rate.

For example, consider mouse movements. In our game, mouse movement directly influences the direction of the player's spaceship. Moreover, the mouse position dictates the mouse cursor position on the screen. From a player's perspective, it would be unacceptable to have observable delay between mouse movements and cursor response. In fact, the player would feel distracted and the game could become unplayable. Therefore, updating the cursor position on the screen should occur as soon as possible and take precedence over other events such as translating mouse movements into changes in ship heading, which could occur after a small delay without game play deteriorating.

Furthermore, other events can be evaluated after longer delays since many game states change slowly with time, if at all. One common example is ship orientation since checking for small deviations in ship position every few milliseconds may be excessive when the processing time could potentially be better spent rendering graphics.

1.2.3 Comprehending Game Time

Thoughtful game design must consider setting the game clock to an actual time and not the processor clock time. A common error in first generation computer games was processing game and graphics logic as fast as the host computer could achieve. Consequently, game speeds (e.g. the speed of moving game objects) widely varied from computer to computer which caused inconsistent game play experiences. In the most extreme cases, faster computers hastened game speed to the point of not responding at all in a timely fashion.

Our team's vision was for game *quality* to vary with computer power, not game *speed*. Thus, a spaceship travelling ten units per second on a first-generation Pentium should travel exactly ten units per second on a new Quad Pentium Core 2 Duo. However, the graphics quality should scale to utilize all available processing power.

To realize the "quality over speed" paradigm with increasing processing power, all game state alterations had to occur relative to a measure of "actual time". Our development strategy was two-fold. First of all, our team changed the game state only at regular periodic intervals defined by actual time (e.g. every 1/60th of a second as opposed to every 400 processor clock cycles). Secondly, every state was conceptually "non-static" with respect to time as described in the sections below.

1.2.4 The Possibility of Timer Overload

In our implementation, "actual time", such as *4:13:01 pm*, is not explicitly used. Instead the function SDL_GetTicks() returns a single long integer, starting from 0, that indicates the number of milliseconds since SDL was initialized. A natural question that arises with this approach is the possibility of the timing variable *overloading* (i.e. could the number of elapsed milliseconds since the game started approach the limit of a long integer's maximal value?).

To answer this question, a short calculation can be performed. A long integer can reach a maximum positive limit of 2,147,483,647 depending on the programming / system environment (up to 18,446,744,073,709,551,615 on some systems).

$$2{,}147{,}483{,}647ms \times \frac{1\text{sec}}{1000ms} \times \frac{1\text{min}}{60\text{sec}} \times \frac{1hr}{60\text{min}} \times \frac{1day}{24hr} \cong 24.9days$$

Therefore, our game would need to run continuously for 24.9 days before overloading of the timing variable would become an issue. Since it's unlikely that any player would ever play continuously for a fraction of this time, overloading the timer is not an issue. Thus, the extra overhead of adding a special check independent of the SDL_GetTicks() function for an overloaded timing variable is unjustified.

However, one proposed strategy for detecting timing overload is to set an SDL timer event for a timing overload check. If an overload is possible within the next period, we could reset the detecting timer to occur at shorter periods, thus checking for timer overload more frequently as the timer variable approaches overload. When an overload is attained and the timing variable rolls over, the programmer would need to evaluate every state at the timing variable's maximum value, and proceed to reset the base time for each game state variable to its rollover value. "Base time" for game states is described in more detail below.

1.2.5 The "Pausing" Conundrum

Because game states are entirely time based, when the game is paused, even if we stop updating the game state variables, time still increments. When the game is unpaused, if no new events are observed, the first game state update will immediately alter all game states to the calculated state for the current time. That is, the game state variables will

discontinuously be assigned to new values as if the pause had never happened.

To correct this issue, a time measurement must be taken when the pause was initiated and when the game is unpaused. The total time the game was paused is then calculable and game state variables can proceed to update continuously accounting for the game state base time value altered by the pause time offset.

1.2.6 Non-Periodic Rendering

As discussed earlier, objects should be rendered to the screen as fast as possible to provide the smoothest animation. Thus, rendering will not occur periodically but instead at irregular intervals whenever the processor is idle. However, rendering the scene depends on the game state. Therefore, to render the current scene, we need to update the game state but, as described above, the game state is updated at periodic intervals. If this logic is correct then it is pointless to render between game state updates, as the same states would be redrawn.

The situation complicates further since the game state is not a collection of static variables that describe instantaneous states but rather game state variables are often function parameters. Using the game state variables, functions can *derive* an instantaneous state at *any* instant in time, which is the key concept of non-periodic rendering.

1.2.7 Contemplating Periodic Rendering

If one examines our team's high-level engine, clearly frames are rendered to the screen only when no other events are pending. Therefore, if several successive events are triggered or if periodic updates occur too frequently or require excessive processing time, then rendering may not occur for lengthy periods of time. Thus, it may be tempting to modify the game engine slightly as described below:

```
While the user is playing and does not wish to exit
{
    Check for input from the user
            Deal with any input quickly
    Check to see if it is time for a periodic update
            If a periodic update is required, deal with the periodic
              update AND render a single frame to the screen.
            If a periodic update is not required, render a frame to
              the screen.
}
```

At surface value this high-level strategy appears to guarantee that the screen would be rendered at least once every period. However, in practice, this rendering tactic backfires badly.

If events or periodic updates are indeed taking too long, thus consuming processing time that could otherwise be used to render graphics, forcing the game engine to render only worsens the situation. While graphics are rendering, additional events and periodic updates are accumulating causing the program to fall behind further.

An optimist may argue that overwhelming the engine with events and updates may only be short term and that the engine will eventually drain the event and update queue and “catch-up” in real-time processing. That is, instead of a rendering delay lasting an entire delay interval, the modified algorithm would provide some interim frames (i.e. valuable player feedback) within the delay interval, at the cost of lengthening the interval itself. Though this scenario may be true one should remember that although objects should be rendered to the screen as frequently as possible for smoothest animation, ultimately rendering is the lowest priority task.

By forcing a render every period, the render will not only increase the interval for which the game engine is overwhelmed (assuming the game engine even can “catch up”) it also delays the execution of *more* important tasks. From the user’s perspective, there is a delayed response observed for each user action, such as delays for the game engine to acknowledge the player has fired a weapon. Even worse, the user could experience a delay between actual mouse movements and observed cursor movements.

Furthermore, there is the possibility that rendering itself takes *longer* than the game update period. In such a case, forcing a render *guarantees* that the game engine will be overwhelmed since during the time to render at least one additional rendering will be queued on the stack. As an analogy, imagine working on an assembly line where outputting an item requires forty seconds, but new items consistently arrive at your assembly station every thirty seconds. In the limit as time approaches infinity, the game engine is guaranteed to be overwhelmed and the “to-render” stack will overflow.

A more palpable alternative is the original strategy of rendering non-periodically, which may skip animation frames but compared to the issues described above, skipped animation frames are an acceptable compromise. Attempting to establish mandatory periodic rendering is actually damaging and potentially crippling to the game engine.

In summary, if the host computer is not rendering graphics because it cannot process updates and events in a timely fashion, then the computer is not powerful enough to play our game and there is no foreseeable change to the game architecture to avoid this shortcoming besides limiting other aspects like 3D model complexity.

1.2.8 Joystick/Gamepad Input Devices

Our team was fully aware during the early stages of game development that the ideal controller would be a joystick / throttle control. A fair criticism of our game is that a specialized input controller is missing from our implementation.

The primary reason for not implementing a joystick-based control scheme is that none of our team members possessed a PC compatible joystick / throttle control, which would severely limit development and testing. A secondary reason for choosing a keyboard and mouse control scheme over a joystick is due to PC game standards and the non-universality of joysticks compared to keyboard and mice (i.e. joysticks can have different numbers of buttons relatively located anywhere in addition to having a throttle control, etc…).

However, conceptually, our application control model facilitates the addition of a joystick with minimal difficulty. To add joystick support several modifications would be necessary including adding joystick-specific events to respond to changes in the joystick state and linking joystick controls to variables and functions that affect the game state. For example, imagine there is a variable called fDistX that determines how fast the player rotates about the local x-axis according to a continuous range from -1 (fast turn left) to 0 (remain straight) to 1 (fast turn right). An input reading from the joystick left/right axis requires normalization to the game state variable fDistX to achieve the desired effect of turning the ship. In particular, the programmer would not be concerned with turning rates, angles, pixels or other internal game engine details. Similarly, alterations to the mouse x-direction and the “right” keyboard button are already mapped in this fashion thus demonstrating the ease of integrating new control schema.

Based on the predicted relative ease of integrating joystick or gamepad support, adding joystick support is a recommended future modification.

1.3 Game Engine Control Logic

Our game engine logic is presented in high-level detail below:

```
While the user is playing and does not wish to exit
{
        Check for input from the user
        {
                If(new user input is detected)
                {
                        Deal with user input quickly
                        If there are events in the queue then:
                                If keyboard events occur, change the appropriate
                                  control variables
                                Else if mouse events occur, change the appropriate
                                  control variables
                        Else if it is time for a periodic update
                        {
                                UpdatePlayer();
                                CollisionDetection(iTimeStamp);
                                AI();
                                SuperAI();
                        {
                }
                Else
                {
                        // No user input is present so render a frame to the screen
                        DrawModel(TimeStamp);
                        DrawAnimations(TimeStamp);
                        DrawDebris(TimeStamp);
                        DrawOverlay();
                }
        }
}
```

Handled immediately

Handled periodically

Handled as often as possible when idle

1.4 Dynamic Motion

For movement calculations, several assumptions were applied to simplify calculations and ease control for the player.

Our team's first assumption was that acceleration is zero (i.e. the player had direct control over both angular and linear velocity). More specifically, the player could control their linear speed and orientation.

The second major assumption is that the player ship velocity vector was always oriented in the current direction the ship was pointed.

It is important to note that the two described assumptions were for the player's movement only. For the enemy ships, velocity is manipulated directly to orient the ships in any provided direction. The simplification for the enemy ships was justified since geometric calculations became easier. From the AI's perspective, calculating final enemy ship heading was simpler than determining the angular speed for which each ship should turn. Thus, enemy ships can change heading instantaneously. However, by a clever use of symmetric models coupled with the fact that the AI determined heading based on the position of the player's ship, which has a finite turning speed, the instantaneous heading change of the enemy ships is predominantly unnoticed.

Thus, from the two assumptions described above, the player ship's instantaneous velocity in the x, y and z directions can be calculated as follows:

1. Use angular velocity to determine the ship's instantaneous orientation.
2. Utilize the ship orientation to calculate a unit vector in the new direction of the ship.
3. Multiply the unit vector by the player ship's scalar speed to calculate the final linear velocities in the x, y and z direction.

However, a first approximation is made to compensate for the limitations of our gaming engine and of a computer with a finite ability to sample mouse positions. In reality, a pilot would continuously vary the ship's control which would result in a constantly changing angular velocity and linear velocity. However, for our game, mouse positions, which are used to calculate angular velocity, are sampled only at discrete time intervals as illustrated in Figure 1 below.

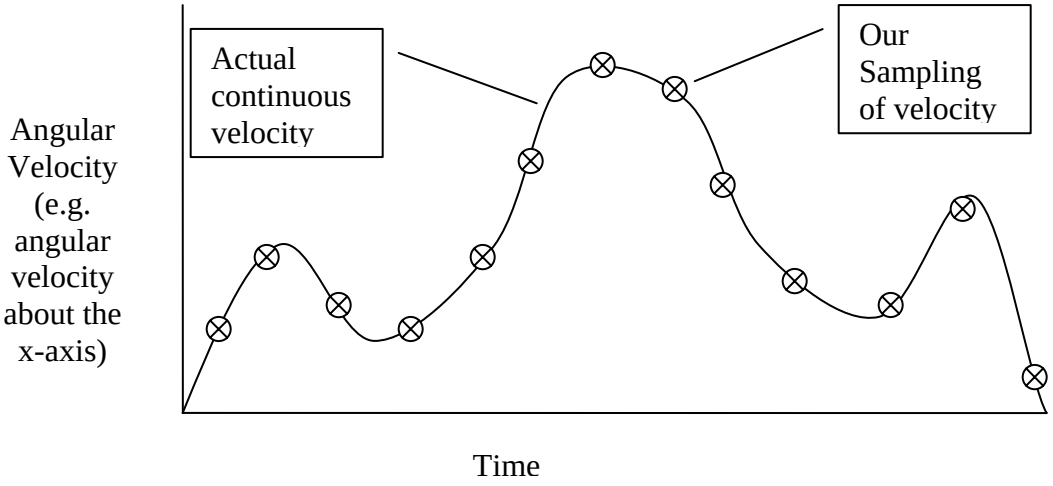


Figure 1. Discretizing angular velocity based on periodic time samples.

Therefore, if the ship's current orientation is required for rendering after sampling angular velocity, we can approximate that angular velocity remains constant between sampling periods as shown in Figure 2 below.

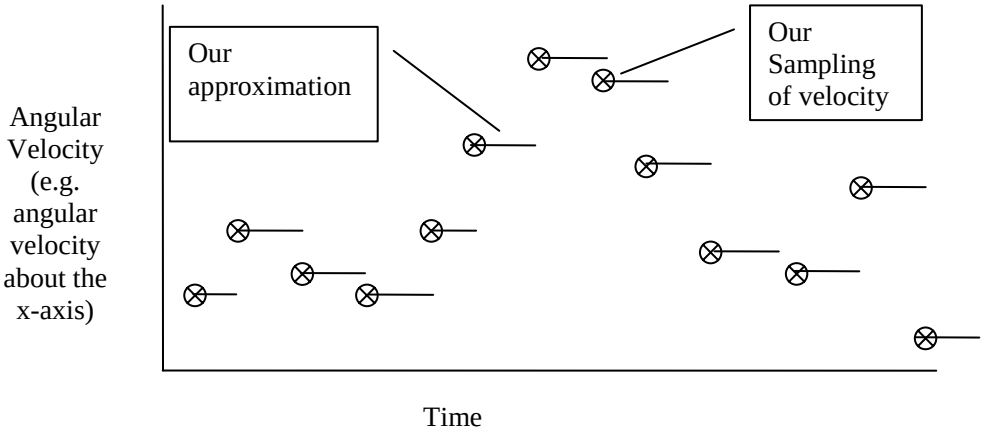


Figure 2. How to approximate angular velocity.

In summary, our first approximation is assuming angular velocity is constant between discrete time intervals.

Note that interpolation between points to obtain more accurate approximations is not used because whenever an angular velocity is calculated, the only known quantity is the last sampled value. Since future samples cannot be predicted interpolation cannot be achieved.

Next, a second approximation is used, namely that linear velocity is constant between time intervals. In essence, the second approximation implies that the player's ship, which would in reality be travelling along smooth curves, is modelled in our game as travelling along short line segments, changing orientation and velocity at discrete time intervals as illustrated in Figure 3 below. However, if the time intervals are short enough, the distinction between the real trajectory and the estimated trajectory is negligible. Since the game's sampling time interval is set to $1/60^{th}$ of second, a linear velocity approximation is acceptable.

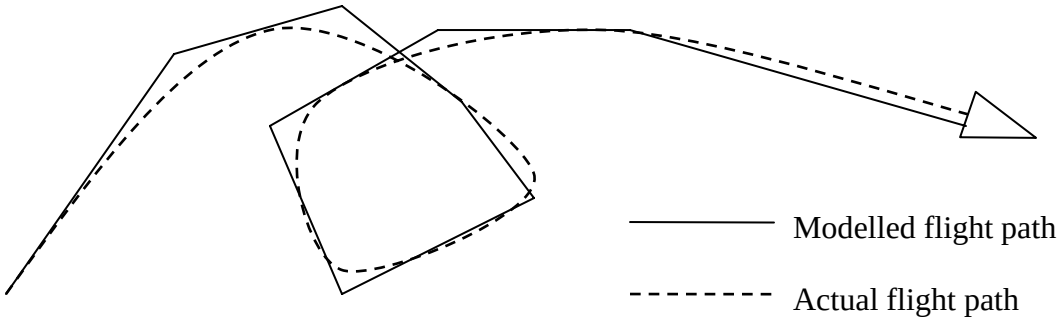


Figure 3. Estimating true player ship trajectory over time.

To calculate the position of any particular object, whether it is the player's ship, an enemy ship, an animation, or anything else in our gaming universe, the following parameters must be known– all of which are stored in the game state variables for each object:

Basetime = the time when all the following parameters were last calculated.

Px, *Py*, *Pz* = the initial x, y, z position at the above Basetime.

Vx, *Vy*, *Vz* = the initial x, y, z velocities at the above Basetime.

Ux, *Uy*, *Uz* = the axis of rotation for a given orientation at the above Basetime.

Theta = the angle of rotation at the above Basetime.

Wx, *Wy*, *Wz* = the axis of rotation for angular velocity at the above Basetime.

WTheta = the angle of rotation for angular velocity at the above Basetime.

Hence, if we need to know the position of any object at any given time in the universe, we can determine the position by the following procedure:

1. Obtain the current time.
2. Subtract the Basetime to calculate a time difference.
3. Multiply the time difference by the velocity.
4. Add the value from step 3 to the initial position.

Similarly, orientation is calculated with a similar principle, though with more complex details as described below. Based on the expressed method, a change in position is not sufficient to warrant a game object

update since position is a derived value. In fact, object updates are only necessary immediately if object linear and angular velocity change.

The game object updating observation described in the previous paragraph reduces processing requirements for movement calculations substantially since objects only require updating when velocities change. Although velocities may alter frequently if the player ship moves continuously, the dozens of enemy ships controlled by a simple AI benefit greatly from this optimization to save valuable processing time.

However, from this description of calculating position, a subtle third approximation was made. Recall that in our game engine we had previously assumed velocity changes with orientation and realized orientation is a function of time. Thus, between game state updates, when the player changes angular velocity parameters the orientation is changing. Therefore, object linear velocity updates should occur between the updates described above. We implicitly assumed between periodic updates that orientation and velocity are independent.

The third approximation is illustrated below in Figure 4. Figure 4 depicts how a ship with constant speed and angular velocity should move between intervals.

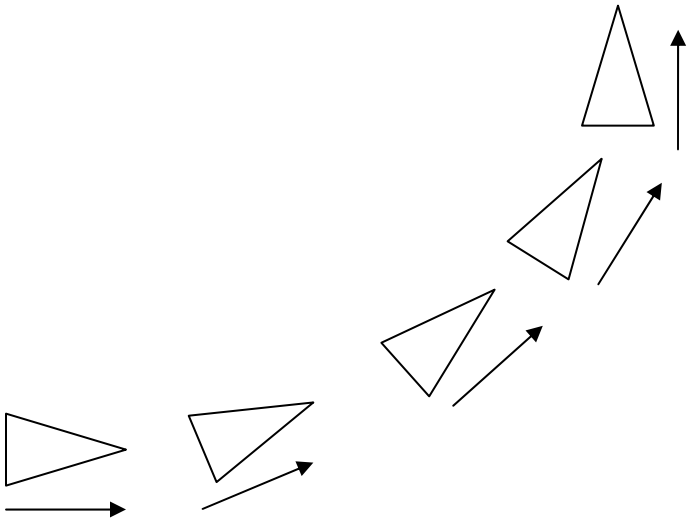


Figure 4. Ideal ship trajectories with constant speed and angular velocity.

However, after applying the third approximation the player ship trajectory closely resembles the schematic in Figure 5 below.

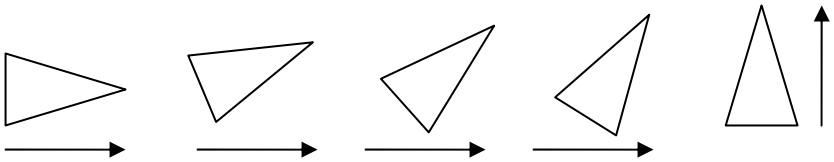


Figure 5. Approximated ship trajectories with constant velocity.

Figure 5 above is exaggerated since it assumes the player ship turns 90º within one periodic update. However, in our game, turns are limited to 3.3º between updates which improves the approximation significantly. In summary, our third major approximation was that between periodic updates linear and angular velocities vary with time independently.

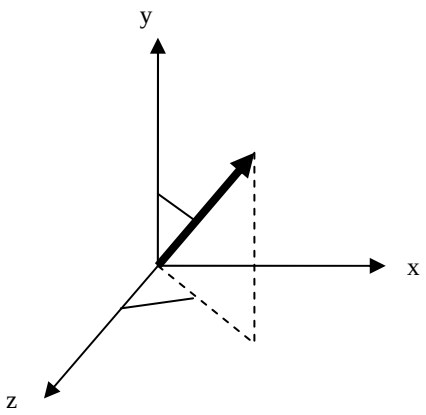
1.5 Orientation

The first step when performing a periodic update and drawing an object is determining object orientation to a good approximation. Therefore, the issue of internally representing orientation is important.

1.5.1 The Euler Angle Misconception

Euler angles are easy to understand. With our original game concept, a gun turret in a fixed position, Euler angles could effectively represent orientation. In fact, for most first person shooter games which involve movement along a ground plane where the “up” vector is relatively fixed, an Euler angle representation works well.

When the user rotates left or right when travelling along the ground plane, the user expects to rotate about the y-axis as shown below in Figure 6. Even if the viewing direction changes, say if the player looks up or down, the user still expects left and right inputs to rotate the player about the y-axis.



Using this simple rotation paradigm for our purposes initially seemed plausible.

In particular, left and right mouse movements could be directly mapped to rotations about the y-axis and vertical mouse movements could be mapped to rotations about the x-axis.

Figure 6. Euler angle rotations.

Thus, a general Euler angle rotation required three separate rotations about each independent coordinate axis. Arbitrarily, our team chose to rotate about the z-axis first, followed by the y-axis and finally the x-axis.

Since gun turrets did not have the “roll” degree of freedom, we could eliminate the initial rotation about the z-axis. Deriving unit vectors from the rotation representing the heading was relatively simple and everything was conceptually consistent.

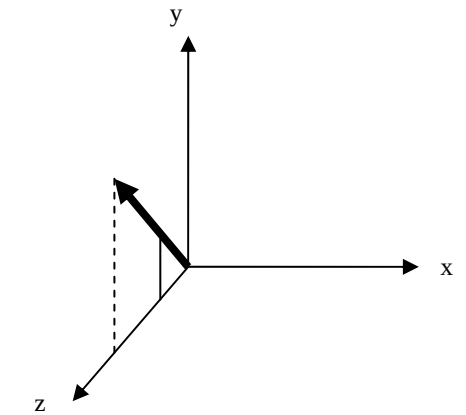
With a gun turret many simplifications were possible. Naively, when our game concept shifted from a player gun turret to a player spaceship we initially decided our orientation and control scheme could trivially continue to be backboned by Euler angles. However, experience taught us a valuable lesson…

1.5.2 Local Coordinate Frames vs. Global Coordinate Frames

Euler angles suffer from a number of drawbacks. One critical drawback is the possibility of Gimbal lock. Gimbal lock occurs when a rotation about an Euler axis rotates an axis of rotation such that the axis of rotation becomes collinear with another axis of rotation which essentially results in a loss of a rotation axis. However, as our control scheme

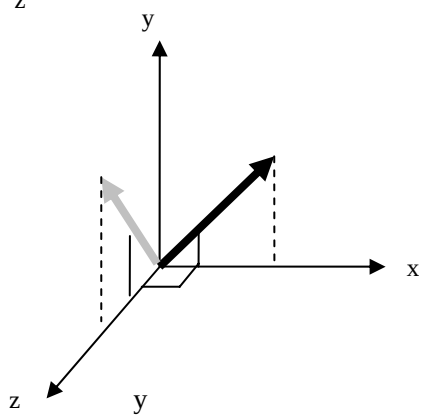
matured, Gimbal lock was not encountered since other issues were more prominent.

The major issues we encountered with rotations were more rudimentary. Namely, given a freely moving and rotating space ship our local and global orientation frames were different. With a gun turret, even a moving turret where the linear frame may change, the orientation frame remains constant. In a space ship, the orientation frame changes constantly. This subtle issue of changing orientation frames is difficult to conceptualize, which partly explains why the problem was not immediately identified. A more detailed schematic explanation is provided below.



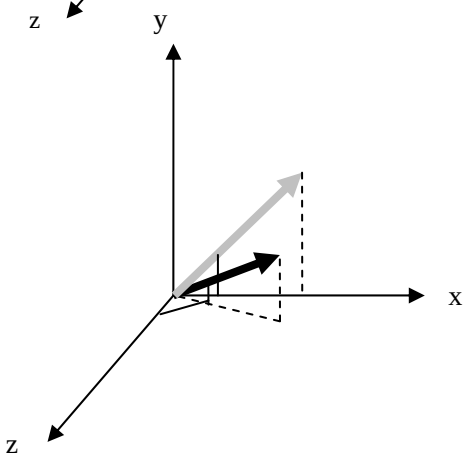
With a gun turret, regardless of player viewing angle, a left or right player input results in a rotation about the y-axis.

As an example, imagine a vector oriented at 45º with respect to the yz plane as shown in the schematic on the left.



If one wants to turn left (i.e. counter-clockwise) by 90º, one would pivot at the origin, parallel to the y-axis, and turn 90º counter-clockwise such that the viewing vector is still oriented at 45º with respect to the z-axis as shown in the schematic on the left.

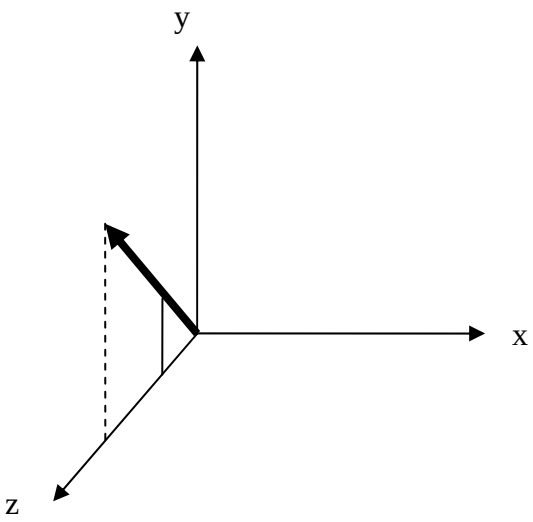
To achieve this result using Euler angle rotations, one could first rotate 45º about the x-axis then rotate 90º about the y-axis without issues.



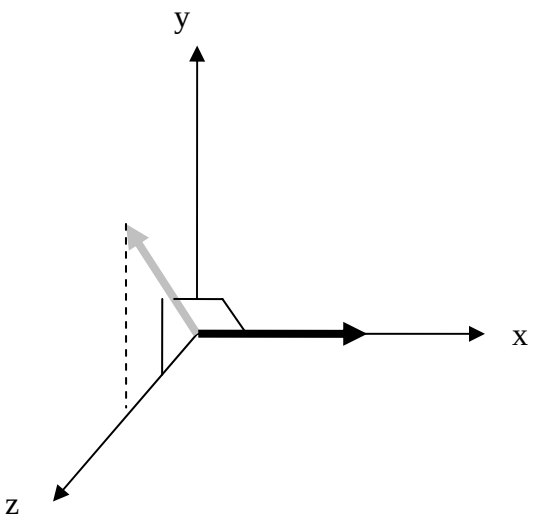
Now imagine one wants to turn right 30º and look down 20º. Therefore, the turning angle is decreased from 90º to 60º, and the up angle changes from 45º to 25º.

Euler rotations by 25º in the up direction then 60º around the y-axis, would result in the expected orientation.

In this case, Euler angles work intuitively because the orientation frame does not move or rotate.

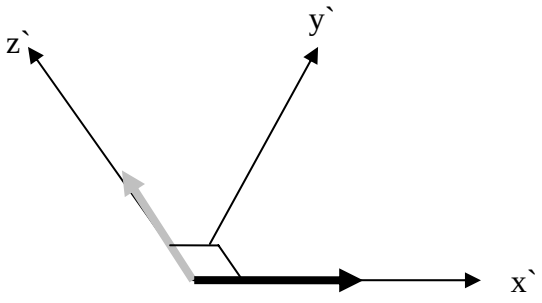


Now, imagine you are a pilot in an airplane angled up at 45º.



Assume the airplane needs to rotate left 90º.

You as the pilot expect to move left relative to your own y axis, relative to yourself not an arbitrary, global, fixed y-axis. A 90º left turn when already facing 45º in the up direction results in a vector positioned along the positive x-axis, which is certainly different from the Euler representation of a 45º turn up followed by a 90º left turn as above.



In summary, as a pilot you would want to rotate 90º about the *local* frame, which is constantly changing based on the current airplane orientation, and *not* the *global* frame.

Thus, “turning left” translates to a different directive for an observer on the ground “looking up” versus a pilot in an airplane oriented at an upward angle.

The described global versus local coordinate frame problem does not obviously manifest itself because rotations purely about the x-axis or y-axis behave correctly since the local frame is changing but subsequent rotations occur about an axis that does not change. For example, if the player is controlling a turret, which is oriented upwards, and the player directs the turret to point further “up” the player would experience the same orientation effect as an airplane pilot pulling up on the control stick incrementally. Similarly, combinations of “left” and “right” turns without intermediate “up” or “down” movement result in correct behaviour for Euler angle representations. The Euler angle problem only becomes evident for combinations of left or right and up or down movements. Even with combined left, right, up and down movements, unless the deviations in different directions are large, the local frame and global frame are similar enough that the difference between the expected response and the actual response is not observable. Hence, the subtle global versus local coordinate frame problem was initially overlooked.

To solve the global versus local coordinate frame problem a mapping must be performed to translate local rotations about the local frame into equivalent rotations about the global frame. However, this problem is geometrically difficult to solve using Euler rotations. Consequently, our team failed to solve the issue.

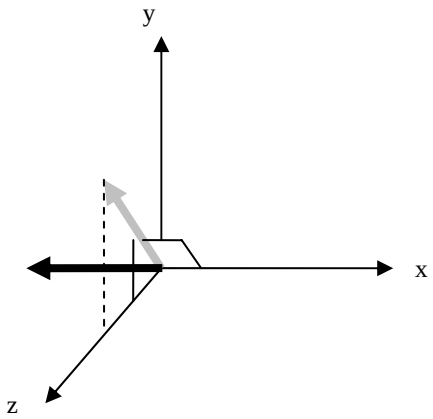
Euler rotations are counter-intuitive at arbitrary angles not aligned with the coordinate axes. In fact, rotations completely “break down” at 90º due to the Gimbal effect which results in “rolling”. At 180º, the controls reverse themselves; for example, moving the ship right actually turns the ship left.

If we were watching the player spaceship from outside, from some fixed camera angle, the system might work since we would once again have a fixed viewing reference, but it would be difficult to accurately judge angles and directions of targets thus straining game play.

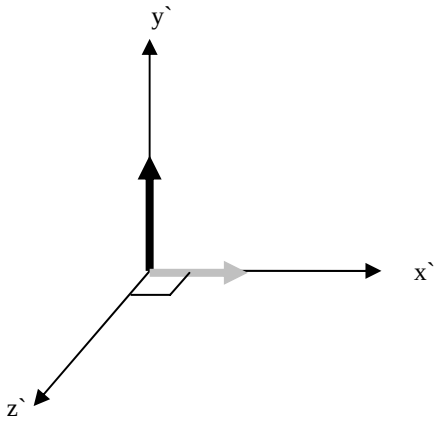
The final implemented solution involved limiting rotations up and down to +/- 45º while allowing full freedom to move right and left. Rolling is disallowed entirely. Though we had a workable control system, rotations up and down would result in increasingly unnatural behaviour. Realistically, spaceships can rotate along any axis at any angle so our implementation is admittedly artificial and potentially frustrating to a human player if an enemy ship is located beyond the imposed 45º limits. Given a second opportunity at implementing the user control interface, quaternions may have been a better option rather than Euler angles.

1.5.3 An Axis Angle Representation

One solution to our control problems is unexpected, and would require us to completely change our representation of orientation. Using an axis angle, instead of some permutation of three rotations about the x, y and z axes, only a single rotation is necessary about some unit vector.



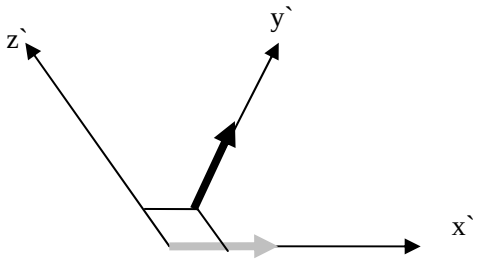
For example, to rotate 45º "up", we would define a unit vector in the $-x$ direction, and a rotation of 45º as illustrated in the diagram to the left. Why should we choose the negative x direction instead of the positive x direction? By convention the right hand rule states if one's thumb is pointed down the axis of rotation, one's fingers curl in the direction of positive rotation. To simplify rotations, our coordinate system is always right-handed.



Next, if we wanted to rotate 90º left about the local axis from the point of view of the space ship the schematic rotation would be that shown in the adjacent figure.

Our axis of rotation is now a unit vector along the positive y-axis (locally) and the angle of rotation is 90º.

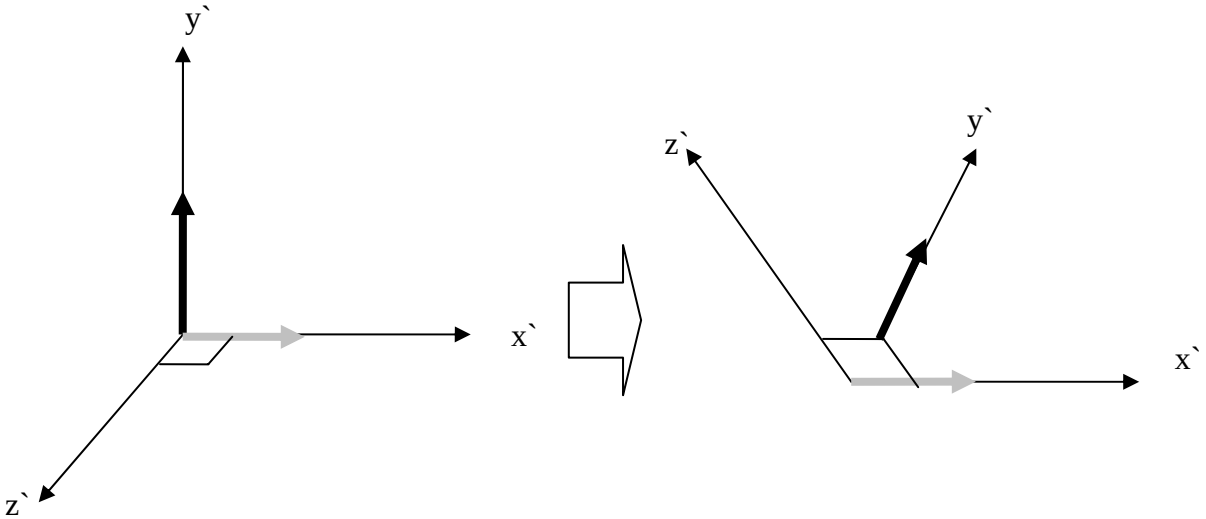
If these two rotations were the only rotations necessary the same local frame issue occurs because we are still implicitly assuming there is a global rotation frame.



Ideally, we should rotate about a unit vector in the y`-direction as shown in the adjacent schematic.

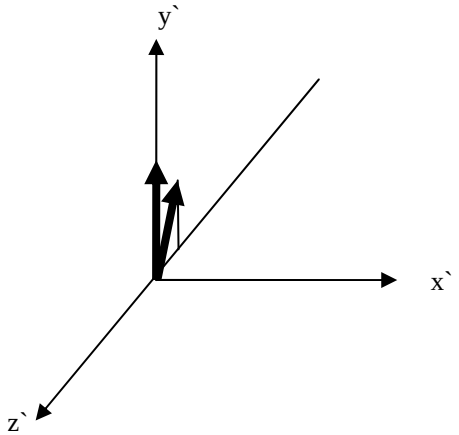
However, if we view the local rotation that the player wishes to execute and observe the rotation in the global frame, we observe that unlike Euler angles, translating a local rotation to a global rotation is easy.

Therefore, the natural question arises: how do we convert from a local angular representation to the desired global angular representation as shown schematically below?



.

Observe that the rotation angle is invariant (for example, 90º in our ongoing discussion). Hence, we need to determine the new rotation axis. If we superimpose the two rotation axes we observe that to translate a rotation in the local frame to a rotation in the global frame, we simply rotate the **local rotation axis** (i.e. the unit vector itself) by the rotation of the local frame. In this example, the local frame was oriented at 45º “upwards” (i.e. a turn about a unit vector in the negative x-direction clockwise).



Hence, our original objective of mapping rotations in the local frame to rotations in the global frame was achieved.

To summarize, performing a rotation about the local frame in terms of a global rotation requires the following steps:

1. Observe the desired local rotation axis R_A described by a unit vector u_A.
2. Rotate the unit vector in step 1 by the rotation of the local frame. Denote this second rotation R_B. Thus, R_B produces a new unit vector u_B.

3. Next, we can combine u_B and R_A to create a new rotation that represents the local rotation in terms of a global rotation. Denote this rotation R_C.
4. Combine the rotation of the frame R_B with the new global rotation, R_C, using matrix multiplication (i.e. $R_C R_B$) to produce the final transformation matrix R_D representing the total rotation with respect to the global frame.
5. Finally, we retrieve the rotation axis and rotation angle from R_D and store them as the new rotation of the local frame (i.e. R_D will be the rotation R_B for the next local rotation).

However, there is still one additional step, namely, how do we translate player input into a local rotation? Using Euler angles, we mentioned that turning left and right meant changes to the angle of rotation about the y-axis, and turning up and down meant changes to the angle of rotation about the x-axis. Therefore, user controls were mapped to angular rotations about the x-axis and y-axis. How would the control scheme change for the axis-angle representation of orientation?

Surprisingly, the transition of the control scheme to an axis-angle representation is straight-forward. Left and right rotations generate a rotation axis in the local y-direction, up and down rotations generate a rotation axis about a vector in the local x-direction, and “rolling” rotations generate a rotation axis in the local z-direction. Therefore, left and right rotations generate a vector in the y-direction, up and down rotations generate a vector in the x-direction and rolling rotations generate a vector in the z-direction. The vector lengths scale with the amount of left/right, up/down and rolling indicated by user input, say by the position of the mouse cursor. Finally, we add each vector together and normalize them to determine our rotation axis. The rotation angle can be set to some constant if a constant rate of rotation is desired. Alternatively, the rotation angle can vary with some scaling factor relative to the length of the pre-normalized

vector sum. There are still issues with an axis-angle representation, specifically singularities at 0º and 180º rotation angles that must be treated as special cases.

Overall, the axis-angle control system permits intuitive, natural movement in any general direction. Calculating directional unit vectors for the axis-angle control scheme is similar to calculating directional unit vectors for the Euler angle representation. As a final bonus with the axis-angle control scheme, Gimbal lock is avoided entirely.

The source code that implements the described axis-angle control system is included below:

```
orientation spaceObject::CurrentOrient (long timestamp)
{
      orientation Jill;
      long TimeDiff = timestamp - basetime;

      //    Interim Values for the calculation
      float RadTheta = theta * M_PI / 180.0;
      //theta in radians
      float c = cos(RadTheta);
            //cos of theta
      float s = sin(RadTheta);
            //sin of theta
      float t = 1.0 - c;
                  //1 - cos theta.
      float x = Ux;
                  //Unit vector normalized to x
      float y = Uy;
                  //Unit vector normalized to y
      float z = Uz;
                  //Unit vector normalized to z

      //First, we establish the initial transformaion matrix based on
the reference rotation axis
      //and angle, i.e. Ux, Uy, Uz and theta.  They have been assigned
above.
      float A11 = t*x*x+c;
      float A12 = t*x*y-z*s;
      float A13 = t*x*z + y*s;
      float A21 = t*x*y + z*s;
      float A22 = t*y*y + c;
      float A23 = t*y*z-x*s;
      float A31 = t*x*z-y*s;
      float A32 = t*y*z + x*s;
      float A33 = t*z*z+c;

      //Second, we need to transform the local frame rotation into a
gloval frame reference.  This
```

```
      //is done by rotating the local rotation axis by the above initial
rotation (in other words, we
      //do a matrix multiplication of the matrix A above with the vector
representing the local
      //rotation axis.
      //Note - Wtheta is a function of time.

      RadTheta = Wtheta * M_PI / 180.0 * TimeDiff;
      c = cos(RadTheta);
                              //cos of theta
      s = sin(RadTheta);
                              //sin of theta
      t = 1.0 - c;
      x = A11*Wx + A12*Wy + A13*Wz; //Unit vector normalized to x
      y = A21*Wx + A22*Wy + A23*Wz; //Unit vector normalized to y
      z = A31*Wx + A32*Wy + A33*Wz; //Unit vector normalized to z

      //Once we have completed the transform, we now find a new rotation
matrix that describes the local
      //rotation in terms of the global frame.

      float B11 = t*x*x + c;
      float B12 = t*x*y - z*s;
      float B13 = t*x*z + y*s;
      float B21 = t*x*y + z*s;
      float B22 = t*y*y + c;
      float B23 = t*y*z - x*s;
      float B31 = t*x*z - y*s;
      float B32 = t*y*z + x*s;
      float B33 = t*z*z + c;

     //Now we multiply the matrixes together to find the final rotation,
this involves rotating about
      //the reference rotation (A) first, and then rotating about the
new rotation matrix (B) second.
      //But in Matrix math, that is actually BA.

      float C11 = B11*A11 + B12*A21 + B13*A31;
      float C12 = B11*A12 + B12*A22 + B13*A23;
      float C13 = B11*A13 + B12*A23 + B13*A33;
      float C21 = B21*A11 + B22*A21 + B23*A31;
      float C22 = B21*A12 + B22*A22 + B23*A32;
      float C23 = B21*A13 + B22*A23 + B23*A33;
      float C31 = B31*A11 + B32*A21 + B33*A31;
      float C32 = B31*A12 + B32*A22 + B33*A32;
      float C33 = B31*A13 + B32*A23 + B33*A33;

      //Now we have the final rotation matrix that describes the total
rotation.

      //Now, we need to convert this rotation matrix back to our axis
angle representation, which is
      //actually a non trivial task...

      //The angle is equal too..
      Jill.theta = acos((C11+C22+C33-1.0)/2.0)/M_PI*180.0;
      if (fabs(Jill.theta) < 0.001)
```

```
	{
		Jill.Ux = 1.0;
		Jill.Uy = 0.0;
		Jill.Uz = 0.0;
		Jill.theta = 0.0;
	}
	else if (fabs(fabs(Jill.theta) - 180.0) < 0.001)
	{
		Jill.Ux = 1.0;
		Jill.Uy = 0.0;
		Jill.Uz = 0.0;
		Jill.theta=180;
	}
	else
	{
		Jill.Ux = (C32 - C23)/sqrt((C32 - C23)*(C32 - C23)+(C13 -
C31)*(C13 - C31)+(C21 - C12)*(C21 - C12));
		Jill.Uy = (C13 - C31)/sqrt((C32 - C23)*(C32 - C23)+(C13 -
C31)*(C13 - C31)+(C21 - C12)*(C21 - C12));
		Jill.Uz = (C21 - C12)/sqrt((C32 - C23)*(C32 - C23)+(C13 -
C31)*(C13 - C31)+(C21 - C12)*(C21 - C12));
	}

	return Jill;
};
```

1.6 Dynamic Game State

Typically, the game state consists of all the variables representing the state of all game objects at a given instant of time. Thus, the game state is the internal representation of the entire game. Example game parameters include real physical quantities such as spaceship positions, velocities and orientations. Moreover, fictitious values like shields, armor and weapons fire are grouped as game state variables. Note that each game state variable is logically dynamic.

However, game state variables are stored statically since variables only change if explicitly updated. For example, if a spaceship position was simply stored at time 0 as (1, 2, 3), and it was not updated until time 5 to (3, 2, 1), then in the interim period between time 0 and time 5, each position reading would be returned as (1, 2, 3). There is clearly an inherent disadvantage of this updating approach. If the screen rendering function was “fast” and could render the game state four times between updates, say at times 1, 2, 3, and 4 then the gained rendering function speed would

be wasted because each graphical position update during these interim times would result in the same (1, 2 ,3) position being rendered.

Instead of storing quantities like position directly, we can store *parameters* such as velocity and reference coordinates such that coupled with time, the position can be derived *dynamically.* With parameter storage, when calculating position, instead of returning a static variable, we can leverage parameters to *derive* the current position. For example, if we store the reference position (1, 2, 3) and velocity (0, 0, 1) at time 0 then at time 5 we can calculate the new position based on the spaceship's last velocity. Hence, each time the rendering function requires a position measurement in the interval time period between game state variable updates, we can use the original reference position and velocity with time and some mapping function to derive a new position.

More concretely if position is equal to reference position + velocity* time then if the rendering function requires position values at times 1, 2, 3, and 4, a new value for position is derived resulting in a different output rendering. Specifically,

At time = 1, we obtain (1, 2, 3) + (0, 0, 1)(1) = (1, 2, 4)
At time = 2, we obtain (1, 2, 3) + (0, 0, 1)(2) = (1, 2, 5)
At time = 3, we obtain (1, 2, 3) + (0, 0, 1)(3) = (1, 2, 6)
At time = 4, we obtain (1, 2, 3) + (0, 0, 1)(4) = (1, 2, 7)

Since the output position value is changing with time smooth animation is observed. In particular, the faster objects are rendered, the smoother the resulting animation. Thus, performance scales with processor power, which is the advantage of a "dynamic game state". At a high-level a dynamic game state is similar to keyframe animation, where only a few animation frames are defined and each interim frame is interpolated from the nearest keyframes.

1.7 Game State Variables

Game state variables define the information required to reconstruct a snapshot of the game state at a particular instant of time. Each game state variable is described independently below.

```
list<spaceObject*>  TheObjList;
```

TheObjList contains every game object that the player can interact with. For example, the player ship, enemy ships and collidable objects are stored in TheObjList.

```
list<AnimationObject*> TheAnimList;
```

TheAnimList stores animations. An animation is defined as user observable events that have no direct influence on other game parameters. Example animations include explosions or weapon effects.

```
list<spaceObject*> WeaponsFireList;
```

The WeaponsFireList contains the results of weapon fire such as projectiles and laser beams. Projectiles, such as missiles, are a “grey area” because projectiles were originally envisioned to be treated as an object in the ObjList until colliding with another object. If a projectile collided with another object, that projectile would be converted into an exploding projectile on the WeaponsFireList.

Given that each variable described above was continuously updated, in addition to which object in TheObjList is the player’s ship and the time for which all the game state variables are valid, we can restore the exact same game state corresponding to the stored time value.

1.8 The Envisioned Multiplayer Network Game

Originally, the game engine was designed with networked multiplayer game play in mind. Classes were designed such that adapting the game engine for multiplayer game play would merely require transporting existing functions to the server-side of the game with minimal modifications. Conceptually, each player client would transmit a periodic update of the player's latest input to the server while the server coordinated game state changes. The logical transition from a single-player game to a multiplayer network game is depicted below in Figure 7.

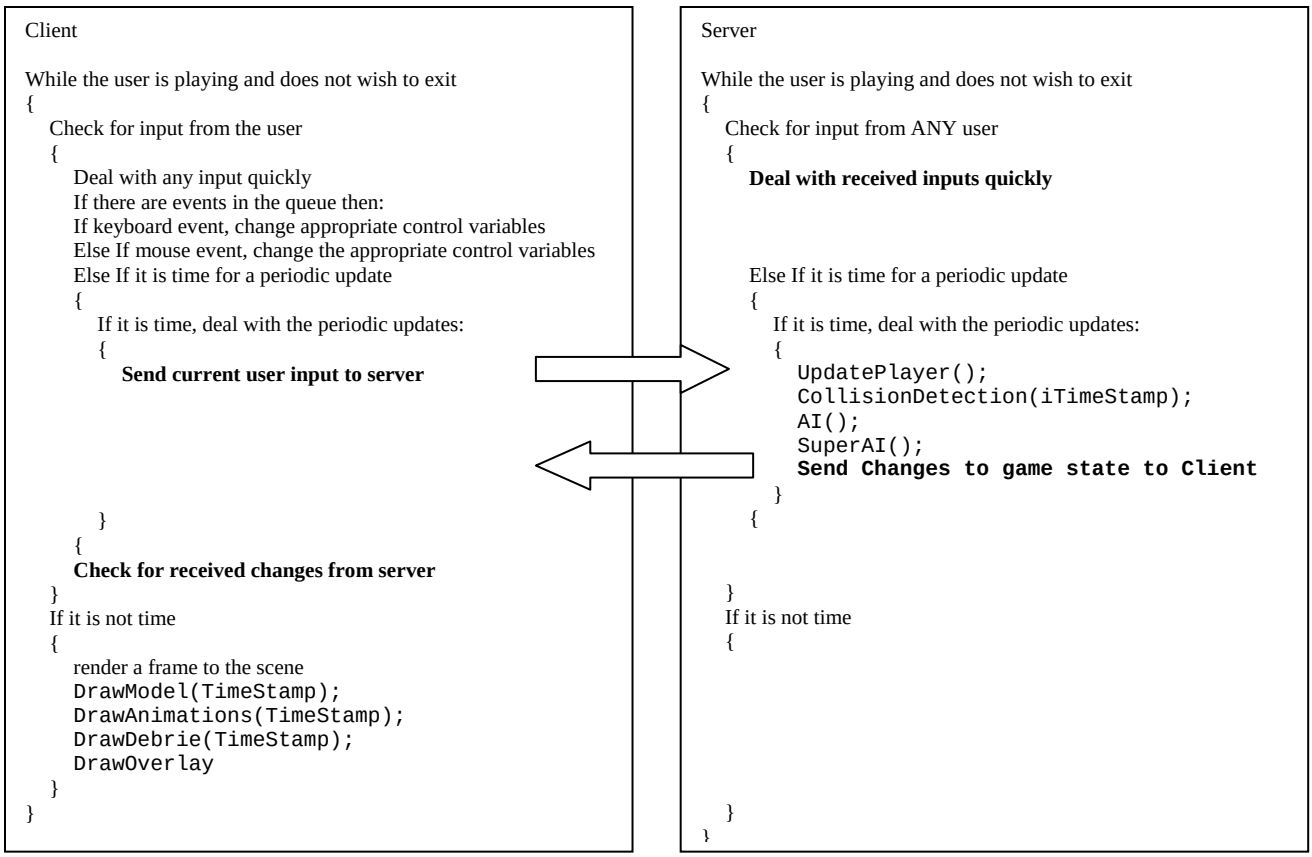


Figure 7. A proposed multiplayer game engine.

1.9 Game Control Mappings

Keyboard Flight Controls

Keyboard Key	Game Control
Up Arrow	Pitch down
Down Arrow	Pitch Up
Left Arrow	Yaw Left
Right Arrow	Yaw Right
,	Roll Left
.	Roll Right
[	1/3 Throttle
]	2/3 Throttle
“	Full Throttle
Backspace	Full Stop
W	Increase Speed
S	Decrease Speed

Camera Controls

Keyboard Key	Game Control
F1	Cockpit View - Forward
F2	Cockpit View – Left
F3	Cockpit View – Right
F4	Cockpit View – Rear
F5	External Chase Camera
F6	External Chase Camera – Left
F7	External Chase Camera – Right
F8	External Chase Camera – Forward
F10	Fixed Camera Position

Other Game Controls

Keyboard Key	Game Control
P	Pause
Esc	Escape

2.0 Game Features

2.1 Game Physics

There are virtually no physics in our game for two reasons:

1. Real physics is counter intuitive to most players as explained below:
    - In real physics, ships could accelerate to near unlimited velocities and would experience inertia when turning.
    - Transitioning to zero velocity using real physics, with freedom of rotation and no friction, is almost impossible since in order to stop moving, one needs to provide a certain acceleration in a certain direction. Any deviations from this acceleration magnitude and direction combination would cause the player ship to move or rotate in some other direction.
    - Most players would likely find movement in one direction while viewing in a different direction counter-intuitive.
    - Most classic space simulation games such as *Wing Commander*, *X-Wing* and *Star Lancer* have fictitious physics.

2. Real physics is not usually required for game play as justified below:
    a. *For collision detection between weaponry and enemy ships or ship-to-ship collisions*: In the case of weaponry and enemy ship collisions, there would be negligible physical effects on the ships, and with ship collisions, like car crashes, the physical collision response (e.g. damped oscillations) is less important than the sheer physical damage caused by the collision.

b. *Newtonian physics*: For the purpose of our game it seemed pointless to assign mass or forces (i.e. thrust) to ships since mass and force are typically employed to derive acceleration or velocity. Therefore, our team opted for the more computationally efficient strategy of working with acceleration and velocity values directly.

The only physics principles present in the game are basic kinematic equations for both linear and angular motion to model object movement as described below:

Position = ½ * acceleration * t^2 + velocity * t + reference position

Orientation = ½ * angular acceleration * t^2 + angular velocity * t + reference angle

2.2 The Super-AI

The Super-AI is the equivalent to the Dungeon Master in Dungeons and Dragons. The Super-AI tracks the overall game state and responds accordingly. Important game state parameters for the Super-AI include the current number of enemies, the elapsed time since the last enemy respawn and the distance between each enemy ship and player ship. Depending on the game state parameters, the Super-AI can dramatically alter the game state. Examples of potential game state changes include introducing new enemy ships, removing enemy ships that have flown out of range and spawning a “boss” ship.

Originally, our team intended to script the Super-AI so levels were fully customizable. However, due to time constraints, the implemented SuperAI is hard-coded in the application. The current logic of the Super-AI

is to spawn random enemy spacecraft formations and have each enemy continuously chase the player ship. When a sufficient number of waves of enemy ships are destroyed, the SuperAI would spawn a boss alien spacecraft for the player ship to engage. While combating the boss alien spacecraft, the SuperAI would introduce additional alien spacecraft that would appear to emanate from the boss spacecraft.

2.3 Enemy AI Behaviour and Logic

The SDL::AI function is found within the SDL_App class of the project. This function controls the behavior of each individual enemy ship in the game world.

The AI behaves as follows: for each shipObject contained in TheObjList (except for the shipObject representing the player's ship), the AI calculates the shipObject's position in the game world as shown in the source code below.

```
TimeDiff = TimeStamp - (*ShipIterator)->basetime;
(*ShipIterator)->basetime = TimeStamp;

x = ((*ShipIterator)->Ax * 0.5f * (GLfloat)TimeDiff *
(GLfloat)TimeDiff) + ((*ShipIterator)->Vx *(GLfloat)TimeDiff) +
((*ShipIterator)->Px);

y = ((*ShipIterator)->Ay * 0.5f * (GLfloat)TimeDiff *
(GLfloat)TimeDiff) + ((*ShipIterator)->Vy *(GLfloat)TimeDiff) +
((*ShipIterator)->Py);

z = ((*ShipIterator)->Az * 0.5f * (GLfloat)TimeDiff *
(GLfloat)TimeDiff) + ((*ShipIterator)->Vz *(GLfloat)TimeDiff) +
((*ShipIterator)->Pz);
```

Overall, the source code snip-it calculates the distance each enemy ship has traveled since the last time the AI function was called. The AI then proceeds to calculate a vector representing the distance between the player's ship and the current enemy ship by taking the difference between the x, y and z positions of both ships. The player-enemy ship vector is converted to a unit vector and the ship's x, y, and z velocities are calculated by simply taking the x, y, and z components of the unit vector and scaling all three results by the speed variable stored in shipObject. Currently, the AI directs every enemy ship spawned in the world towards the player's current position. Once an enemyShip's position and direction has been adjusted, the ship is sent to the ObjRenderList for rendering.

Originally, the AI was prototyped to be a simple, 'proof-of-concept' design to be enhanced as game development progressed. However, due to time constraints, the simple AI concept has minimally evolved to become the main enemy AI. While the "player ship chasing" AI may seem overwhelming since enemy ships will constantly adjust their heading towards the player, the player's ship has a greater maximum speed than the enemy ships permitting the player to "escape" undesirable enemy confrontations before collision.

The AI function is run every periodically from within the TimerEventHandler function.

2.4 Ship Collision Detection and Logic

The SDL_App::ShipCollision function determines if any enemy ships have collided with the player's ship and is also responsible for the game logic behind ship collisions.

The player's ship and enemy ships are assigned a 'size' variable which represents the minimum radius of a sphere which bounds the ship in question. The collision detection algorithm is simple; SDL_App::ShipCollision calculates the distance between an enemy ship and the player's ship. If this distance is less than the added radii of the

enemy ship in question and the player's ship then a collision has occurred. ShipCollision then proceeds to execute the game logic behind collisions. The calculations involved in finding the distance between two ships utilize the same equations as used in the SDL_App:AI function; ShipCollision finds the enemy ship's current position and subtracts this position from the player ship's position. The magnitude of this vector is the distance between the two ships.

When an enemy ship collides with the player's ship, ShipCollision applies damage calculations; if the player's ship has less than zero hit points after the collision, the player ship is destroyed and the SDL_App::GameOver function is called. If the player survives the collision, an appropriate number of hit points are subtracted from the player ship's hit points and the enemy ship involved in the collision is destroyed and removed from the game world.

SDL_App::ShipCollision is called periodically from the TimerEventHandler through SDL_App::CollisionDetection.

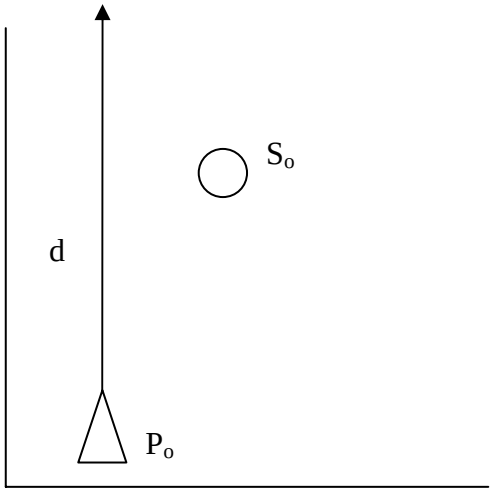
Given additional time to optimize the AI, our team would have set limitations on the enemy ships' angular velocities, which would have effectively given the player the option of performing turning manoeuvres rather than relying on the player's greater maximum speed. Moreover, we could have implemented an enemy ship AI that flees from the player ship once its healthy decreases below a certain threshold.

2.5 Weapon Collision Detection

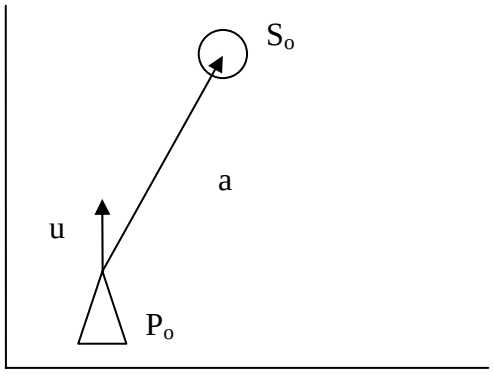
2.5.1 Instant Hit/Miss Weaponry

The weapon collision detection algorithm is slightly more complex than the bounding sphere collision detection algorithm used for ship collisions. Weapon collision detection is performed once for every potential target and we can instantly calculate if a weapon fired collides with a target

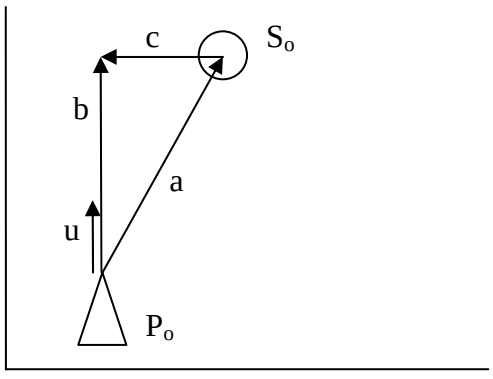
or not. The net effect is exactly analogous to using bounding sphere game objects. An example weapon collision calculation is explained below.



Suppose we have a space ship at P_o and a target at S_o. Moreover, suppose the spaceship at P_o fires a laser beam *d*, which is represented as a directional vector, as depicted in the adjacent diagram.



First of all, we calculate the unit vector *u* representing the direction of *d*. Another vector, *a*, defines the trajectory from the player ship to the target ship.



Next, we calculate the projection of *a* on *u*, which is the vector *b*. Subtracting *a* from *b* results in the vector *c* as illustrated in the adjacent diagram.

The length of *c* indicates the minimum distance between the laser beam and the target. If the length of *c* is less than the target size, there is a *potential* hit. The hit is only *potential* because the above calculation only determines whether a hit occurs assuming the laser beam is an infinite line in both directions. Two further checks are necessary to establish if the physical laser beam "hit" the object in question. The first check is ensuring that vector *b* is positive so that only objects in front of the ship are hittable while the second check is determining if the length of *b* is less than the maximum range of the fired weapon.

The complete source code for weapon collision detection is provided below.

```
void SDL_App::WeaponCollision(long TimeStamp)
{
	position ImDoomPosition;
	position ImTargetPosition;
	list<spaceObject*>::iterator ShipIterator;
	list<spaceObject*>::iterator WeaponIterator;
	float Ux, Uy, Uz;                                  //Unit vectors
representing direction of ship
	float Bx, By, Bz;
	float Cx, Cy, Cz, Cdot;
	float n;                                                  //Dummy
Parameter for parameterization
	float InterimX, InterimY, InterimZ;       //Interim Holders
	float DistanceToDeath;                             //How far away
from beam
	int hitTrue = 0;

	//Only do collision detection if the ship is running
	if (PlayerShip != NULL)
	{
		//We must test for each weapon fire in the universe
		for (WeaponIterator = WeaponsFireList.begin();
WeaponIterator != WeaponsFireList.end(); )
		{
			hitTrue = 0;

			ImDoomPosition = (*WeaponIterator)-
>CurrentPosition(TimeStamp);

			//Get the unit vector of the laser beam
			Uy = -sin( ImDoomPosition.Pa / 180 * M_PI );
			Ux = fabs(cos(ImDoomPosition.Pa / 180.0f * M_PI)) *
sin(ImDoomPosition.Pb / 180.0f * M_PI);
			Uz = fabs(cos(ImDoomPosition.Pa / 180.0f * M_PI)) *
cos(ImDoomPosition.Pb / 180.0f * M_PI);
```

```
                        //printf ("\nWeapon(start): Px: %f, Py: %f,
Pz: %f\n", ImDoomPosition.Px, ImDoomPosition.Py, ImDoomPosition.Pz);
                        //printf ("Weapon(end): Ux: %f, Uy: %f,
Uz: %f\n", Ux, Uy, Uz);

                  //We must test for teach object in the universe EXCEPT
the player's ship
                  for (ShipIterator = TheObjList.begin();
ShipIterator != TheObjList.end();)
                  {
                        if ((*ShipIterator) != PlayerShip)
                        {
                              ImTargetPosition = (*ShipIterator)-
>CurrentPosition(TimeStamp);

                              //Get a vector from the beam to the origin
of the laser
                              Bx = ImTargetPosition.Px -
ImDoomPosition.Px;
                              By = ImTargetPosition.Py -
ImDoomPosition.Py;
                              Bz = ImTargetPosition.Pz -
ImDoomPosition.Pz;

                              //Get the projection of the beam to the
ship
                              Cdot = Bx*Ux + By*Uy + Bz*Uz;

                              Cx = Cdot * Ux;
                              Cy = Cdot * Uy;
                              Cz = Cdot * Uz;

                              //Get the shortest distance to the beam
                              DistanceToDeath = (Cx-Bx)*(Cx-Bx) + (Cy-
By)*(Cy-By) + (Cz-Bz)*(Cz-Bz);

                              if (DistanceToDeath <= 4.0)
                              {
…
```

2.5.2 Projectile Weaponry

Although the algorithm for projectile collision detection is straight-forward, there were prohibitive reasons why projectiles were not present in our game. The general algorithm for projectile collision detection is for every game state update, test to determine if the projectile is within the bounding sphere radius of any ship. If the projectile is within the bounding sphere radius then a hit occurs. Clearly, the projectile collision detection

algorithm is the same algorithm as ship collision algorithm; the algorithmic complexity is not the issue.

The underlying problem stems from the number of algorithm executions. Each projectile requires the algorithm to test for collisions against every object in the game during each periodic update. Moreover, each projectile position must be continuously tracked. The processing requirements increase exponentially with each projectile fired. Thus, for a game with multiple human players or several enemies simultaneously firing projectiles, processing requirements are prohibitively expensive.

To circumvent the processing issue, one can impose a limit on the number of active projectiles, a common tactic in video games. However, limiting the number of projectiles results in an artificial game where weapon rates of fire are slow. In particular, firing an automatic projectile weapon would feel unrealistic to most players.

However, projectile weapons fire can be *simulated*. A simple simulation would reuse the collision detection algorithm for laser beams a single time for each projectile to determine if the projectile would hit the target or not and then animate a travelling projectile.

2.6 Creation of 3D Studio Max Models

Our space shooting game requires basic models, such as the player space ship, enemy alien ships and potentially other space objects. Creating models to represent space objects is tricky using primitive OpenGL shapes, especially for individuals with a limited artistic background. Moreover, a substantial amount of optimized rendering code would be required in OpenGL drawing functions, likely using display lists, in addition to extra overhead required to effectively track objects. Using 3D Studio Max, models can be created efficiently. Our team used 3D Studio Max as it is an industry standard so experience gained can be transferred to the 3D Studio Max expert’s resume. Moreover, there are open-source 3D Studio Max loaders widely available for use.

The first implemented model was the player's spaceship. The player spaceship design is loosely based on the *Star Wars* X-Wing fighter as shown in Figure 8 below. Using primitive shapes such as pyramids, pipes, and ovals the basic player ship shape could be crafted. In general, 3DS Max provides a sleek interface for the Cartesian camera angles which permits intuitive 3D manipulation of the drawn model as exemplified in Figure 9 below.

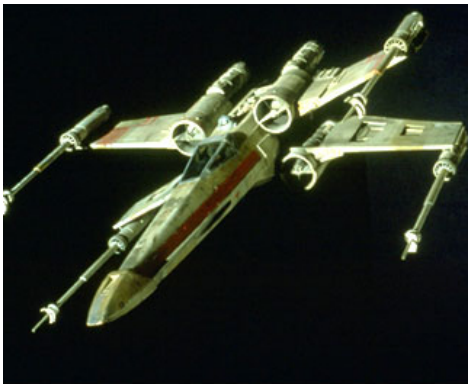


Figure 8. An X-Wing fighter: the basis for the player ship.

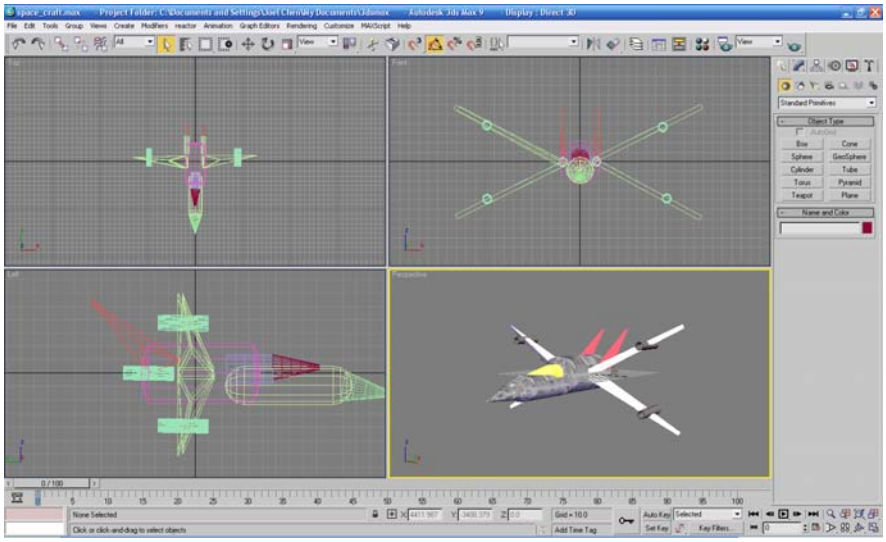


Figure 9. 3D Studio Max design interface.

After designing the foundational model shape, a 2D texture can be applied using the 3DS Max material editor. The material editor allows users to modify the color of the ship as well as add 2D textures to the ship's hull. Each texture can be imported as a JPEG image file. The texture file used for the ship's hull is shown below in Figure 10.

Figure 10. Player ship hull texture.

Proprietary 3D Studio Max files can be saved in the 3DS format for later importing into an OpenGL application using a 3DS loader. Leveraging these fundamental model creation techniques, other models were produced for the enemy alien ships (UFOs), missiles, and the alien mothership as illustrated below in Figures 11, 12 and 13.

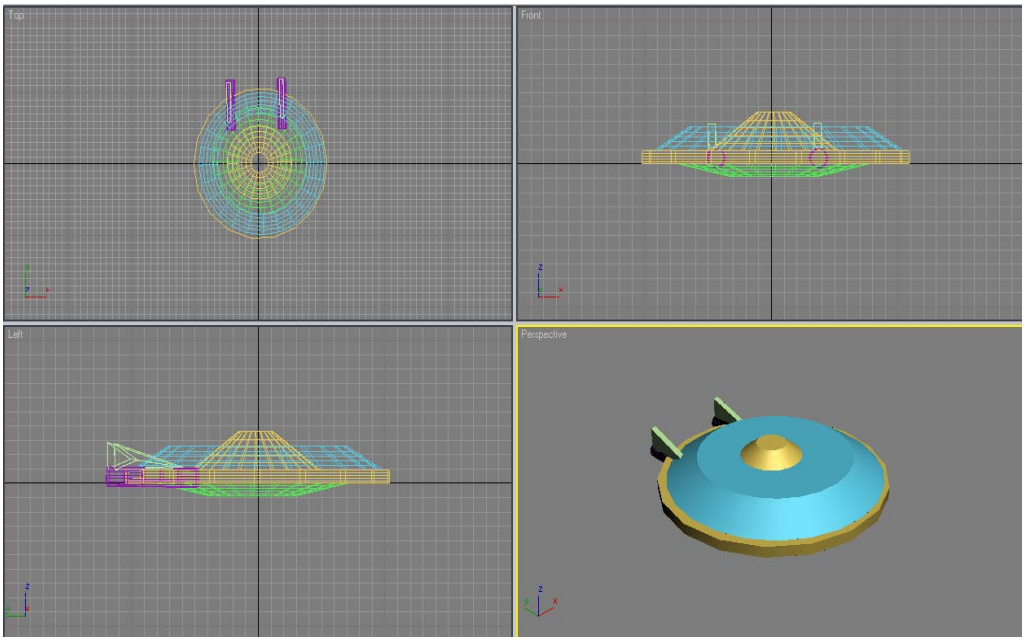


Figure 11. 3D Studio Max alien ship (UFO) model.

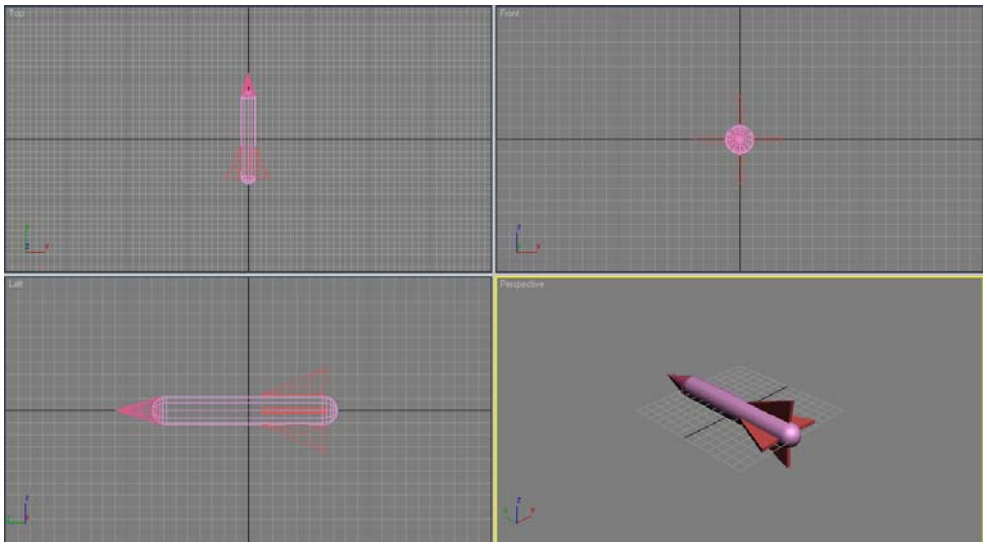


Figure 12. 3D Studio Max missile model.

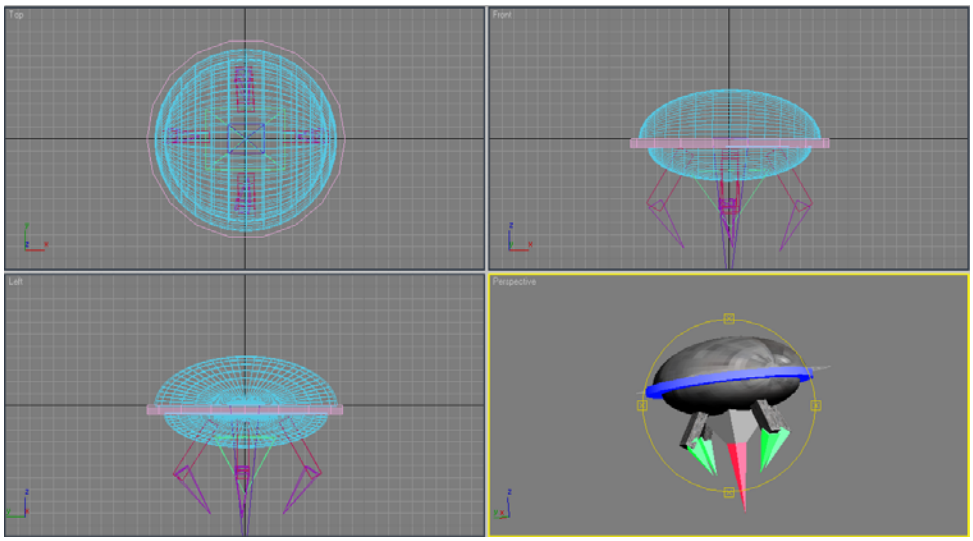


Figure 13. 3D Studio Max alien mothership model.

At this stage, space objects needed to be loaded and rendered in our OpenGL game environment using a 3D Studio Max loader, as described in the next section.

2.7 3D Studio Max Model Classes

Having created models in 3D Studio Max some method was needed to load and render each model. Our team opted to wrap the open-source library Lib3ds (http://lib3ds.sourceforge.net/). The Lib3ds API provides a mechanism to load and render 3D Studio Max files, as shown in Figure 14 below; however, the core API does not offer texture mapping support. To support texture mapping for imported 3D Studio Max models, the Lib3ds API had to be integrated into our main game project and abstracted as a class, namely the SDL_3DSModel class.

To integrate Lib3ds into our game project, our team linked into the Lib3ds library through Visual Studios, which was straightforward given the Lib3ds DLL files. The SDL_3DSModel class is adapted from Lib3ds sample code which loads a 3DS file into a GLUT window and suggests a framework for texture mapping.

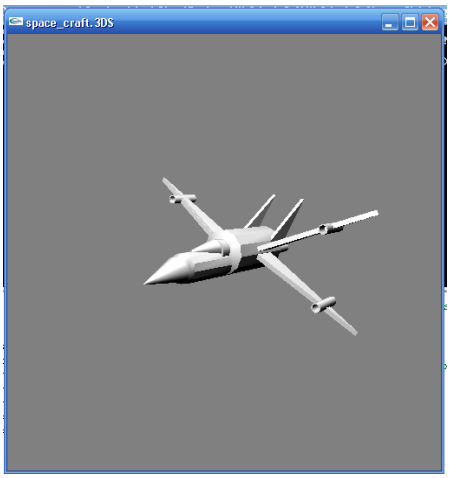


Figure 14. 3DS Loader rendering the player ship model.

To read JPEG image texture maps another third-party library, called SDL_Image, was leveraged. Source code from the open-source SDL_Image library was incorporated in the SDL_3DSModel class. In general, the SDL_3DSModel class abstracts a model originally created in 3D Studio Max. The parameterized class constructor has a filename input for creating a tree data structure representing the model. Whenever the

model requires rendering, a call to the public function “draw” will recursively render the nodes for the model. Within the class implementation, textures mapped to exported 3D Studio Max models are properly rendered with the SDL_Image library with the caveat that lower resolution textures should be used to minimize processing time. For example, the enemy ship hull texture is only 100x100 pixels to optimize performance on standard machines as exemplified in Figure 15 below.

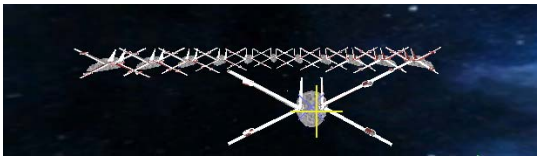


Figure 15. Texture-mapped player ship rendered in the game.

Overall, the SDL_3DSModel class provides an efficient encapsulation for loading 3DS model files into the game.

2.8 The Heads-up Display

For the player to effectively pilot the spaceship and be aware of his/her surroundings, a heads-up display (HUD) is necessary. The heads-up display presents the current player ship status including health and score. Moreover, the HUD provides a three-dimensional radar to determine where the enemies are located in space.

The HUD module is abstracted from the main animation class. In particular, the HUD class contains the rendering code for the health bar, player score and 3D radar as shown in Figure 16 below.

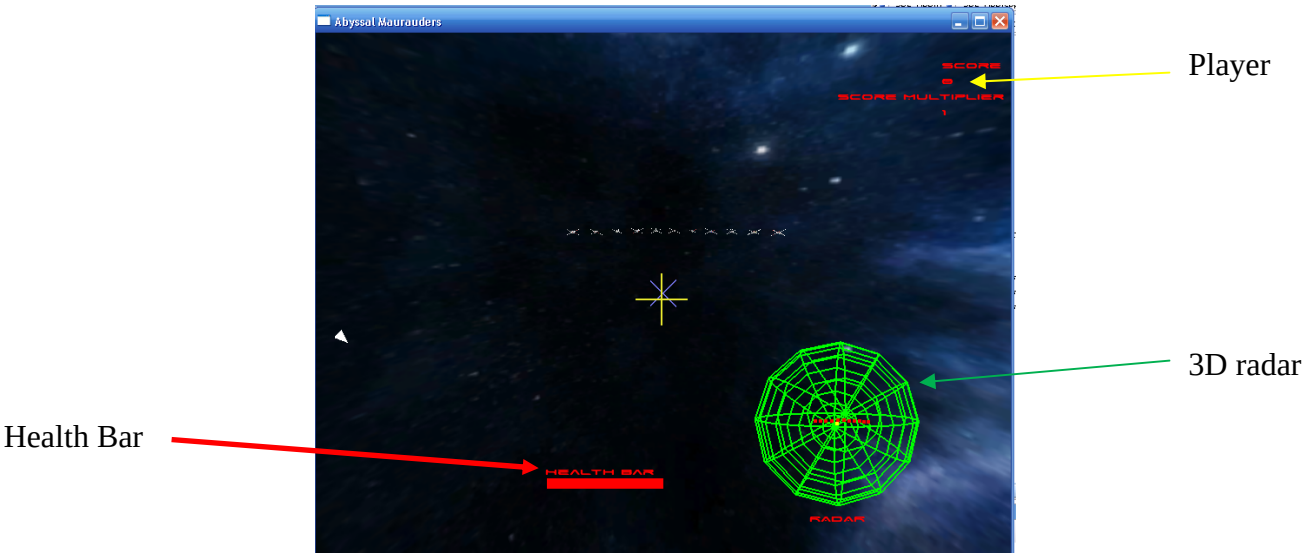


Figure 16. The player's heads-up display.

Each component of the HUD is rendered in a separate viewport in OpenGL to maintain a separate projection matrix and perspective from the main rendering code. In particular, the 3D radar viewport occupies a quarter of the screen (i.e. the bottom right-hand corner) while the health bar is rendered in the bottom middle of the screen.

The HUD constructor requires the height and width of the SDL application to properly size the viewports for display and if the SDL window is resized, the viewport dimensions must be updated. The health bar is drawn using a 2D rectangle with the rectangle length scaled according to the player's current health. The HUD text, including labels and score, is implemented using the 2D Text class, as described in the 2D Text section.

Drawing the 3D radar requires rendering a green, 3D wireframe gluSphere without lighting. The radar sphere is rendered at the center of the viewport with the gluLookAt function looking at the center of the sphere. A list of spaceObjects is passed into the drawRadar function to determine which space objects should be visible on the radar. When the radar is drawn the positions of each space object in the world are processed by calculating their distance with respect to the player's ship and comparing if

the object is within the visibility texture-mapped sphere. If the object is within the texture-mapped sphere, a red point is rendered inside the wireframe radar sphere to represent the space object. Moreover, the radar rotates as the spaceship is rotating. The current angle of the radar is adjustable by invoking the rotateRadar function which rotates according to Euler angles.

Overall, the heads-up display (HUD) class creates a simple, intuitive display for the player and provides an efficient means to render the HUD by higher-level callers.

2.9 Rendering 2D Text to the Screen

To display static and dynamic information to the user, primarily in the HUD (Head's-Up-Display), a text display mechanism was required. Static information included labels for the health bar and radar in addition to a "Game Over" message while dynamic information included the player's current score and score multiplier. To effectively render two-dimensional text to the screen for the user's benefit, several third-party libraries were investigated. The obvious option was to investigate existing two-dimensional font implementations from well-known, OpenGL, open-source websites, such as NeHe Productions (http://nehe.gamedev.net/). Although NeHe Productions presented a detailed tutorial for rendering two-dimensional text to the screen, the source code was predominantly Windows specific so extracting the critical tutorial information to create an API for text rendering was deemed time-consuming and inefficient by our team.

An alternative approach to text rendering was to employ an open-source, lightweight API named glFont developed by BYU student Brad Fish (http://students.cs.byu.edu/~bfish/glfont.php). Though the provided wrapper for the FreeType rasterizer seemed promising as it could theoretically render any TrueType font to the display, glFont usage was rejected

because the sample source code did not execute as advertised. Thus, the final decision was to wrap the open-source, SDL TTF API (available at http://www.libsdl.org/projects/SDL_ttf/) using a namespace interface rather than a class interface for ease of use. The wrapper created for the SDL TTF library is shown below:

```
namespace SDL_2D_Text
{
void InitTTFLibrary();

void loadTTFFont(const char * const fontFileNameAndPath, int
pointSize, int specialFontFormatting, TTF_Font *&font);

GLuint create2DTextTexture( TTF_Font * font, const char *const
message, SDL_Color * foregroundColour, int& textureWidth, int&
textureHeight, GLfloat textureCoordinates[]);

void draw2DText(GLuint textTexture, int textTextureWidth, int
textTextureHeight, int lowerLeftTextXPosition, int
lowerLeftTextYPosition, GLfloat textureCoordinates[]);

void delete2DText(GLuint textTexture);
}
```

From a client programmer's perspective, to render text to the screen the namespace functions InitTTFLibrary()and loadTTF() must be called in succession to initialize the TTF library and then load the specified TTF file with a given point size and special formatting (e.g. bold, underline or italized characteristics). Next, the client programmer passes a message and desired text foreground for the newly loaded font to the create2DTextTexture(). The create2DTextTexture() function returns an

integer representing a handle to the texture loaded in GPU memory for the provided message string. To transform the user message into a texture, the text is first rendered to a SDL surface before being passed to a texture creation function. Finally, the original message is rendered to the screen as a texture mapped to a rectangle at the provided lowerLeftTextXPosition and lowerLeftTextYPosition by invoking draw2DText(). Text textures can be deleted from GPU memory by invoking the delete2DText() function on a integer handle referencing the text texture.

Tests utilizing the SDL_2D_Text namespace functions suggested the original library and wrapper worked exactly as expected and in particular the rendered text can be easily scaled and repositioned for screen resizing operations. Overall, the SDL_2D_Text namespace provides a convenient, extensible set of functions to interact with two-dimensional text and is leveraged throughout the HUD. An example of how two-dimensional text is used in the HUD is shown below in Figure 17.

Figure 17. 2D Text in the HUD.

2.10 Scoring System Mechanics and Logic

A scoring system is included in the game logic to add a general game objective and improve game replayability. Whenever the player destroys an enemy ship with the ship's weapons, a certain number of points are added to the player's score. The number of points awarded is based on an enemy ship's base score multiplied by a scoring multiplier variable. The scoring variable increases if the player manages to hit enemy ships in succession without missing a single shot thus discouraging players from adopting a 'spray-and-pray' mentality with their weapons handling. Shown in Figure 18 below is a game screenshot with the player's current score and score multiplier shown in the top-right-hand corner of the screen.

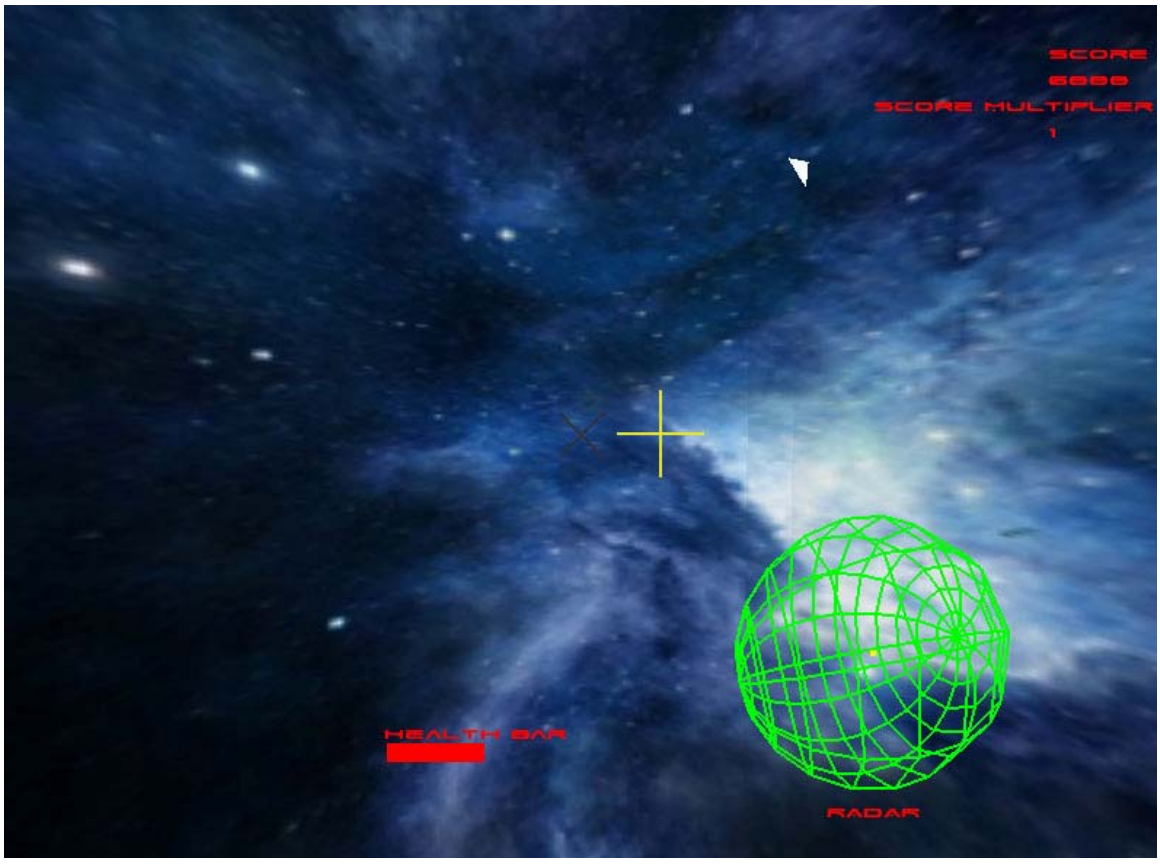


Figure 18. Score with multiplier.

The scoring system is currently embedded inside the SDL_App::WeaponCollision function since the scoring system is directly dependant on if the player hits or misses with their weapons.

2.11 Animation

Animations are an integral component to any game as user feedback is provided when an event occurs. For Abyssal Marauders in particular, the three principle animations are: lasers, for damaging enemy spacecraft; explosions, which occur when an enemy spacecraft is destroyed; and enemy ship damage, which occurs when an enemy ship is damaged but not destroyed.

Our game architecture includes a base class for all animations called the animationObject class. The animationObject class inherits from the spaceObject class thus information regarding spaceObject position, velocity, and Basetime when the object was created is known. An animation class adds a draw function, which renders a frame of the animation, and returns a 1 when the animation is completed. Since the animationObject class is an abstract class, application code can store an array of animationObject pointers for different types of animation.

Each principle type of animation is contained within its own class. The laser animation occurs when the player presses the mouse to fire the laser weapon as shown below in Figures 19 and 20. Currently, the lasers are simply glLines with a line width of twenty. Since the player ship is modeled after a *Star Wars* X-Wing fighter, four laser beams are emitted with every weapons fire. The laser object must have a copy of the player ship rotation angle to ensure the lasers are pointed in the right direction using rotations about the Euler angles. Finally, the laser speeds are initialized to 0.01 and double every rendering cycle to provide the illusion that the lasers are travelling near the speed of light. A laser sound effect is created within the AnimLaser class by calling the playSound() function. A

potential improvement for the laser animation is adding a glow texture to provide an overall science-fiction feeling to the game.

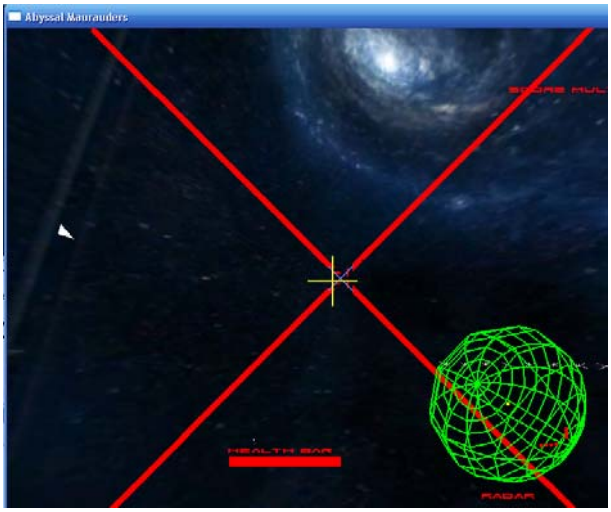


Figure 19. First-person perspective of a laser animation.

Figure 20. Third-person perspective of a laser animation.

The explosion animation occurs when an enemy ship is destroyed by the player's weapons. The explosion animation is implemented using triangle particles, which act as debris, and emit at random speeds and rotations creating the illusion of shattered glass as shown in Figure 21 below. For each enemy ship explosion, the current position of an enemy

spacecraft is calculated and the explosion animation is rendered until one second of game time has elapsed. If time permits 3D textures can be added to each explosion particle to provide a fire effect. Moving the *s*, *t* and *u* vectors of the 3D texture can provide additional animation to the explosion.

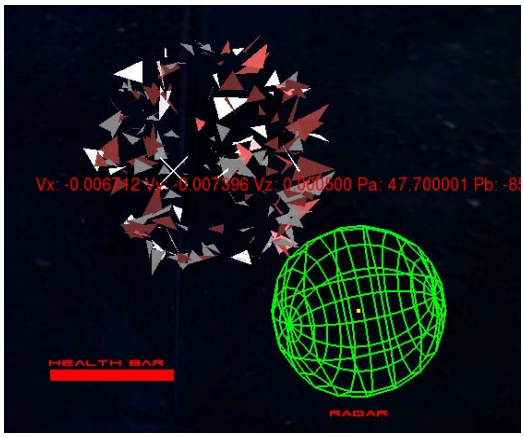


Figure 21. Explosion animation.

The enemy damage animation occurs when a player weapon collides with an enemy spacecraft but does not destroy it. Since a shield on the enemy spacecraft prevents it from being destroyed, a shield animation should be rendered. A growing gluSphere encapsulating the enemy spacecraft implements the shield animation. One trick employed to render the enemy shield was to use alpha blending so the enemy shields appeared transparent as shown in Figure 22 below. In effect, the shield animation provides user feedback to signify if the player weapon collided with the enemy spacecraft.

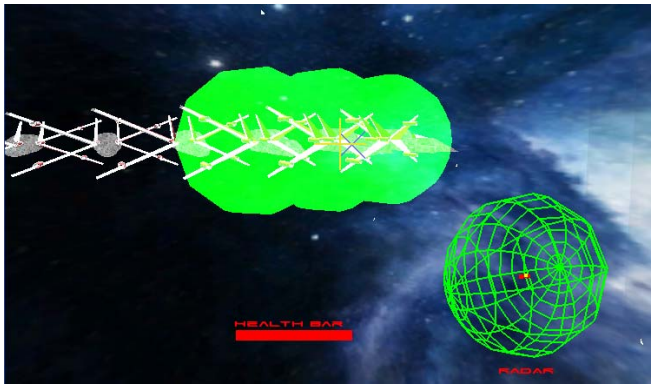


Figure 22. Enemy spacecraft shield animation.

2.12 Sphere Texture Mapping for the Space Background

To improve game realism and professionalism our team decided a space background was necessary. The objective of the background was to move with respect to the spaceship and permit the player to rotate their ship 360º along any of the three coordinate axes to see different regions of space. To achieve this effect, a space *.TGA image was loaded into memory and subsequently texture mapped to a sphere.

The first stage of texture mapping to a sphere was loading the texture into memory. Invoking loadTGAfile(), which requires a texture file path and name and a integer to represent a handle to the texture file loaded to memory, uploaded the texture to GPU memory. Each invocation of loadTGAfile() attempts to open the specified *.TGA image file and examine the file's metadata, including image resolution and color scheme, then load the image into texture memory by calling the glTexImage2D() OpenGL function. However, before glTexImage2D() is called, the texture image properties are set for repeating in the *s* and *t* directions with linear interpolation for the MIN_FILTER and MAG_FILTER. The only restriction with the loadTGAfile() function is that the image resolution had to be a power of two in the x-direction and y-direction to a maximum of 512 x 512 pixels and only *.TGA image files could be loaded into memory.

Drawing and texture mapping the outside of a sphere was achieved by incrementally texture mapping wedges at the sphere surface by varying the lines of latitude and longitude in a nested for loop. For visually-appealing texture mapping results that minimally affected performance, the sphere was divided into thirty-six lines of latitude and seventy-two lines of longitude. During each frame render, the sphere was redrawn and texture mapped centered at the location of the player's ship with a radius of sixty units so that the player's line of sight was minimally constrained.

Despite the general successes associated with sphere texture mapping, there were two unresolved difficulties. The first issue was texture

wrapping which caused discontinuities at the boundary where the sphere was first texture mapped as shown in Figure 23 below.

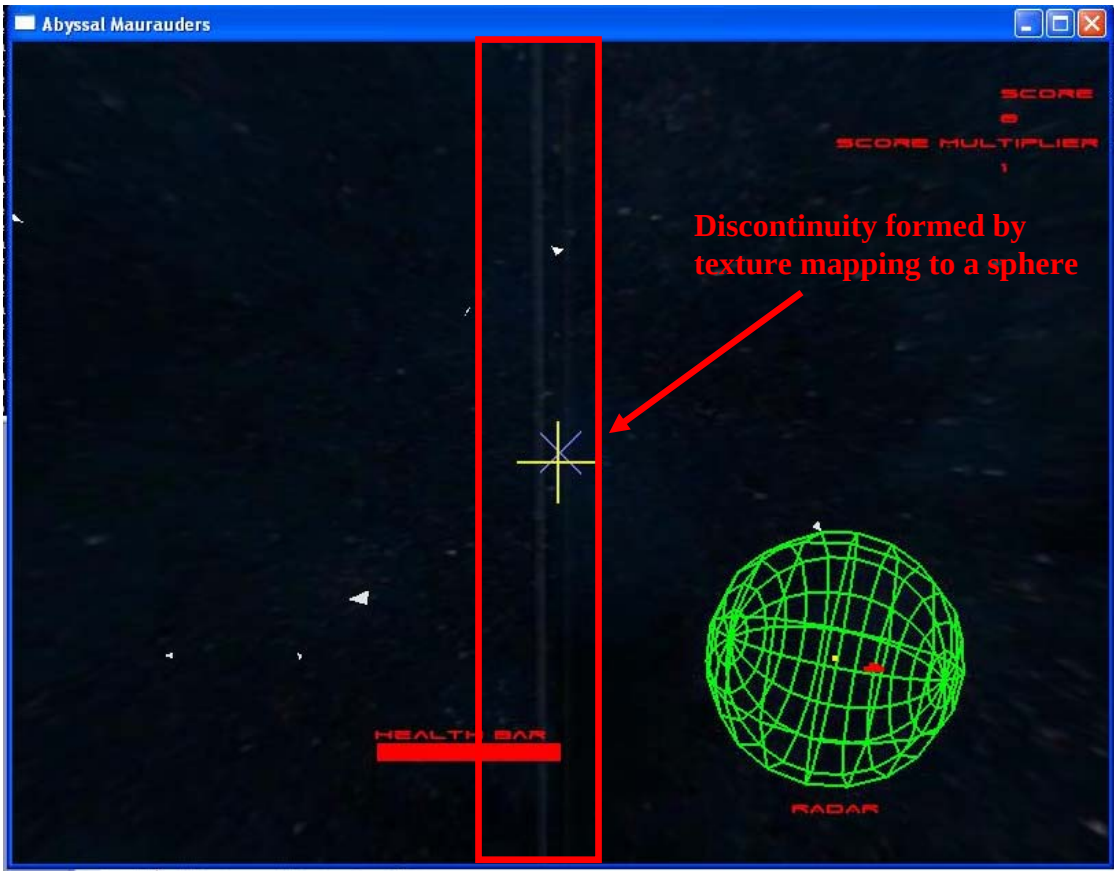


Figure 23. Discontinuity created by sphere texture mapping.

A natural solution to the sphere texture wrapping problem would be leveraging a high-resolution, tileable image of space to ensure wrapping the image would naturally not cause discontinuities. However, tileable high-resolution images of space were difficult to acquire from the Internet.

A second issue with texture mapping was distortion and limitations for image resolution. Due to the texture loader employed, a loaded texture could have a maximum image resolution of 512 x 512 pixels and each resolution dimension had to be a power of two, primarily due to restrictions of the OpenGL glTexImage2D() loader. Therefore, the sphere texture required disproportionate resizing before loading into the GPU thus creating distortion.

2.13 Game Menus

The game menu enables users to restart a new game, alter game parameters and pause the game. A game menu system can accommodate non-real-time settings. For example, game commands can be remapped through an options menu. The game menu's implementation is primarily accomplished with 2D texture mapping and 2D text textures. In particular, 24-bit Windows bitmap files are used for different game menu screens. Each bitmap file was generated from edited artwork by resizing the images and modifying the commands with Adobe Photoshop CS3 to fulfill our particular game requirements. For example, the original background image for the main game menu is shown below in Figure 24 and the same image overlaid with our custom game menus is shown below in Figure 25.

Figure 24. The original background image for the main game menu.

For the menu page bitmap files, we added text for each menu page with different titles. All the added text is given specific effects such as shadow, inner glow (specularity), outer glow (emissivity) and color gradients to enhance readability and the aesthetic appearance of the menu page. Most of these operations were accomplished by using blending properties and layers in Adobe Photoshop CS3. All of the generated bitmaps were also resized to 1024x512 pixels since the SDL_LoadBMP() function only works with dimensions that are powers of two.

Figure 25. The original background image overlaid with custom game menus.

The menu page bitmaps are loaded when the menu is initialized or when the application is initialized. Once loaded into an array in memory, the bitmap images can be reused for future invocations of the game menus. However, the 24-bit Windows bitmap files are saved in a characteristic BGR color format instead of the standard RGB. Consequently, calling SDL_LoadBMP() will result in correct texture mapping with the wrong color scheme. To solve this issue, the texture data had to be converted to a suitable format to ensure expected texture mapping and colorization as exemplified in the code fragment below:

```
if ( TextureImage->format->BitsPerPixel == 24 )
   {
       printf("Handle 24 bits");
       TextureSize = 3 * TextureImage->w * TextureImage->h;
       TextureData = (unsigned char*)TextureImage->pixels;

       for (int Pointer  = 0; Pointer < TextureSize; Pointer+=3)
       {
           Temporary = TextureData[Pointer];
           TextureData[Pointer] = TextureData[Pointer + 2];
           TextureData[Pointer + 2] = Temporary;
       }
   }
```

An example of an incorrectly loaded game menu bitmap is shown in Figure 26 below.

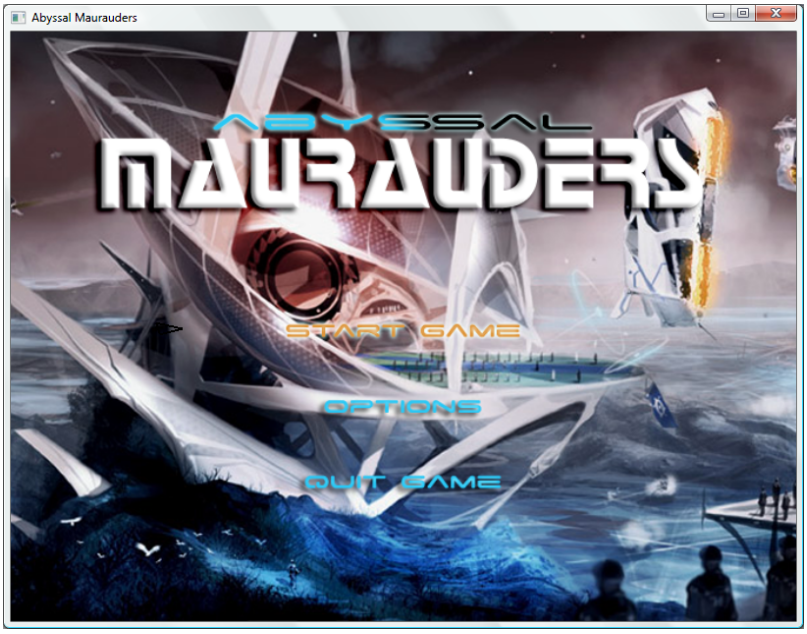


Figure 26. An incorrectly loaded 24-bit *.BMP file.

Moreover, our team implemented a credits scene, which is integrated into the game menu, by 2D texture mapping. For the game credits, the text is predominantly generated using a 2D text texture map rather than Adobe Photoshop text. Alpha bending is utilized extensively in the game credits.

To smooth transitions between the game state, pause menu state, main menu state, and ending scene state, the architecture shown in Figure 27 below was implemented.

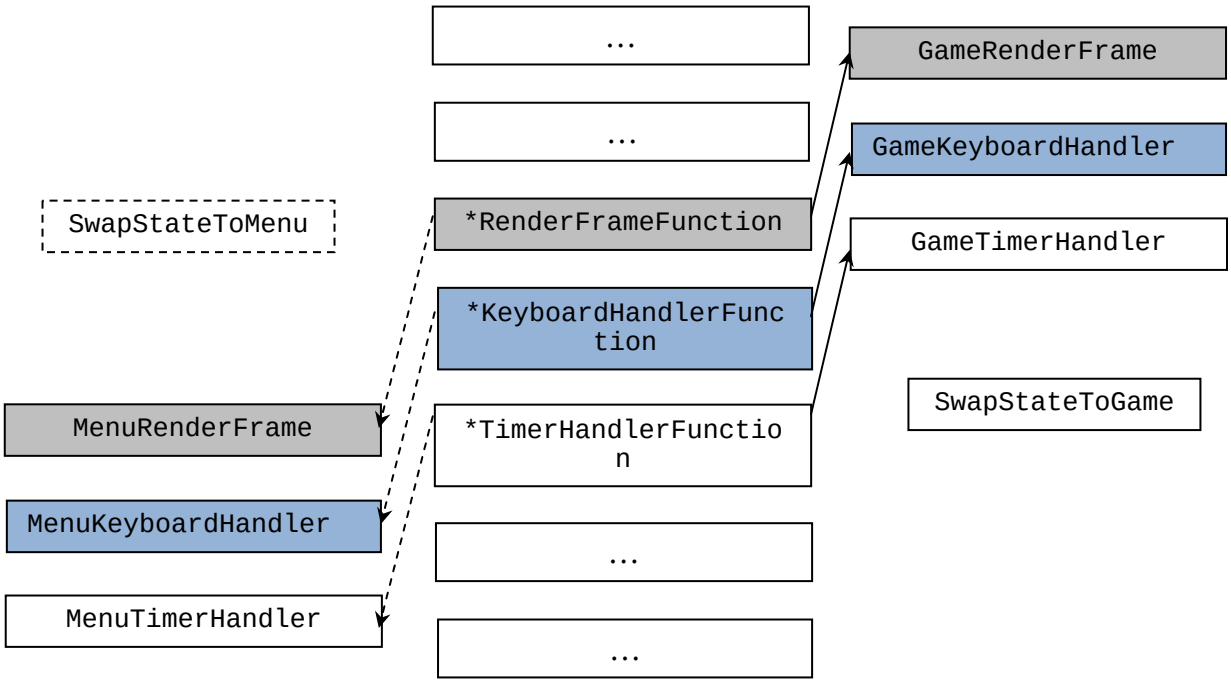


Figure 27. Menu and game callback interaction architecture.

When the application is initialized, the user observes the game menu state. Throughout the application, the SDL event handler manages user input, timers and screen rendering by invoking the appropriate callback functions. For example, the event handler calls the RenderFrame() function when the screen requires updating. Moreover, user keyboard input is handled by a KeyboardHandler() function. Similarly, timer events trigger TimerHandler functions to update game state variables. When a new game is started, the SwapStateToGame() function is invoked and all the function pointers described in the menu and game callback interaction architecture diagram above will point to the GameRenderFrame(), GameKeyboardHandler(), and GameTimerHandler() functions. Thus, the application's behavior can seamlessly transition between the main game, main menu, pause menu, and ending scene states.

Based on the state architecture described above, the game is paused by invoking the SwapStateToPauseMenu() function, which removes event handling for the GameRenderFrame() and GameTimerHandler() functions. Consequently, the game state at the time of pausing is preserved until the user exits the paused game state by invoking SwapStateToGame(). With this architecture each logical game state can have its own distinct input handlers. For example, pressing the “enter” button on the menu will not cause the player’s spacecraft to release a bomb and menu navigation will not alter the player spacecraft’s orientation.

Currently, the game menu implementation is embedded within the main SDL_App class. However, the game and menus do not logically share information so abstracting the game and menus as different classes would be a reasonable idea except that the function pointer scheme illustrated in Figure 27 above is infeasible with a class separation. Moreover, by embedding the game menu logic within the main SDL_App class significant overhead is avoided by invoking additional functions with parameters that would require checking.

2.14 Sound and Music

In video games, sound effects and music are considered significant features as audio additions enhance game play. Sound effects and music complement the visual graphics by providing companion sound effects for different events including collisions and explosions. If implemented well sound assists the player in responding to changes in the environment. Moreover, background music can increase the player’s level of excitement by engaging the player’s sense of hearing to accompany stunning visual effects. For these reasons, our team opted to incorporate both sound effects and music into our game.

At the game's level of abstraction, audio is handled by a standard SDL library that monitors sound-based events. As a result, the audio feature inherits the portability aspects associated with the SDL library implementation so thus is compatible with other platforms such as Windows, MacOS and UNIX.

In the rudimentary SDL library, the most common implementation pattern is the following logical sequence:

1. Initialize the SDL Audio.
2. Implement a sound sample buffer.
3. Specify the desired audio format.
4. Create or obtain a *.WAV file for playback, possibly using an audio format converter.
5. Load the *.WAV file(s) into an audio buffer.

Next, an audio callback function will continuously cycle through the audio buffer for playable sounds and potentially playback multiple audio files simultaneously. Therefore, the audio buffer size will impact the response time and drop rate of the audio playback. Once a sound sample in the buffer has finished playing, the sample is removed by a new loaded *.WAV file. For example, if a siren, gunshot, and scream sound are played during game execution, each *.WAV file will be loaded, converted, and stored in an available slot in the sound buffer. Each sound will be played when the audio callback function iterates through the sound's buffer slot. A diagram illustrating the audio playback process is shown below in Figure 28.

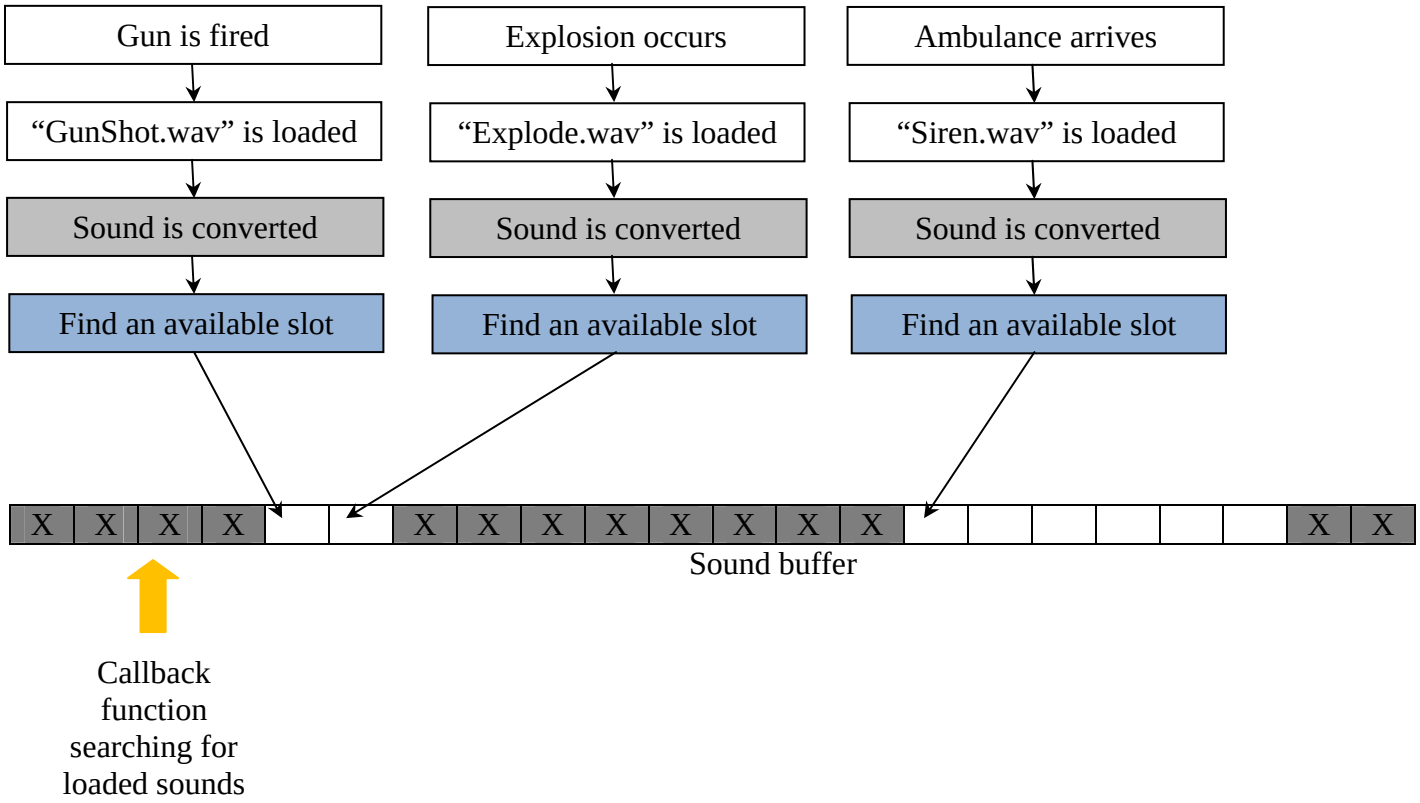


Figure 28. The SDL audio playback process.

Originally, the sound interface for Abyssal Marauders was implemented as described above since this implementation permitted audio playback at any time within the game by simply invoking the PlaySound() function. Any audio file encoded in the *.WAV file format was playable. However, the SDL library lacked the ability to loop music and mix audio files to different volume levels. Moreover, the PlaySound() function inefficiently loaded the same *.WAV file multiple times if multiple playbacks of the same sound were required. To overcome these limitations and improve audio performance other audio API options were investigated.

After researching several third-party SDL-compatible audio APIs, our team opted for the SDL plug-in library named SDL_mixer, which is implemented by the team who implemented the SDL library. The SDL_mixer plug-in library provides the features missing from the standard SDL audio library. Thus, music and sound effects can be mixed separately permitting substantial flexibility and control. Furthermore, sound effects can

be logically grouped together enhancing the sound experience. Additionally, SDL_mixer loads audio files to dynamic memory so each sound clip can be replayed anytime in the game without reloading the file. Most importantly, music can be easily looped and faded without extra event timers and handlers. A final feature of SDL_mixer is MP3 decoding support through the external mpeg.dll library.

Despite the benefits offered by the SDL_mixer library, conflicts arise with the native SDL audio functions already leveraged extensively in our audio interface. However, the majority of the audio invocations are implemented similarly in SDL_mixer with added features thus mixing function calls between the two APIs could result in conflicting behavior. Although our team had a working audio interface to satisfy the fundamental game requirements, SDL_mixer was selected for its extra features and efficient implementation.

In summary, the SDL_mixer command sequence can be described by the following steps:

1. Initialize the SDL audio mixing by specifying all possible audio playback formats.
2. Instantiate a separate sound sample and music buffer for each mix.
3. Load all sound and music files into a buffer.
4. Playback the music and sound files at appropriate points in the game.
5. Close the audio stream and free dynamic memory allocated before terminating the application.

The final audio API functions are shown below:

```
void PlaySound(SoundSamples Sample);
void PlayMusic(MusicSamples Sample);
void InitializeGameAudio();
void CloseGameAudio();
```

Both PlaySound() and PlayMusic() are callable throughout the game provided the InitializeGameAudio() function is invoked during game initialization. Furthermore, both PlaySound() and PlayMusic() require an enumerated parameter to specify the desired sound effects or music such as a LASER or EXPLOSION. The enumerated parameter abstraction allows developers to conveniently change the audio clip filename without modifying each invocation of PlaySound() or PlayMusic() that plays that particular audio clip. CloseGameAudio() must be invoked when the application terminates to free all reserved memory resources.

The current game audio abstraction implementation plays a single sound for each PlaySound() function call. However, most commercial video games support an audio architecture such that a single triggered sound event, such as an explosion, can trigger other sound events simultaneously which results in a rich and realistic sound experience. Nevertheless, our game's audio API supports multiple sound events by successive calls to the PlaySound() function. Error-checking can be performed during game initialization to ensure all required audio files are present in the game folder hierarchy. If files are missing, the application can terminate immediately.

2.15 Software Configuration Environment

To create an intuitive software development environment where different files could co-exist within a logical directory hierarchy several measures were taken. To fully comprehend these measures, a background of common Visual Studios 2005 elements is required since Visual Studios 2005 was the primary software development tool employed for all game project development. The following description assumes the reader has some experience using Visual Studios 2005.

For the game development environment, the main solution file is located in ~\EECE_478_Game_Project. The Visual Studios 2005 solution file encapsulates the main game project file. An important distinction when

building a project is realizing if the release configuration or debug configuration is being built. Ideally, the debug and release builds should complete successfully since debug builds permit the programmer to effectively troubleshoot by setting breakpoints at the cost of extra files generated and larger file sizes while release builds act as a final copy of the software with more compact files. The developed game supported both debug and release build configurations.

For individual source file compilations within the EECE_478_Game_Project project, warnings will be displayed. Most of the warnings stem from outdated function calls from third-party library code so thus can safely be ignored. However, the programmer should be aware of header file and library dependencies for third party, open-source implementations which could cause issues with compilation and project linking.

By right-clicking on the project within Visual Studios, choosing “Properties” then looking under “Config Properties”→ “C/C++” → “General” → “Additional Include Directories”, one can view the list of third-party header file inclusions. To determine how each macro is being resolved (e.g. what the macro “SDL_HOME” is resolving to for the main game project), left-click the “Additional Include Directories” field then click the “…” button → “Macros <<” as shown in Figure 29 below.

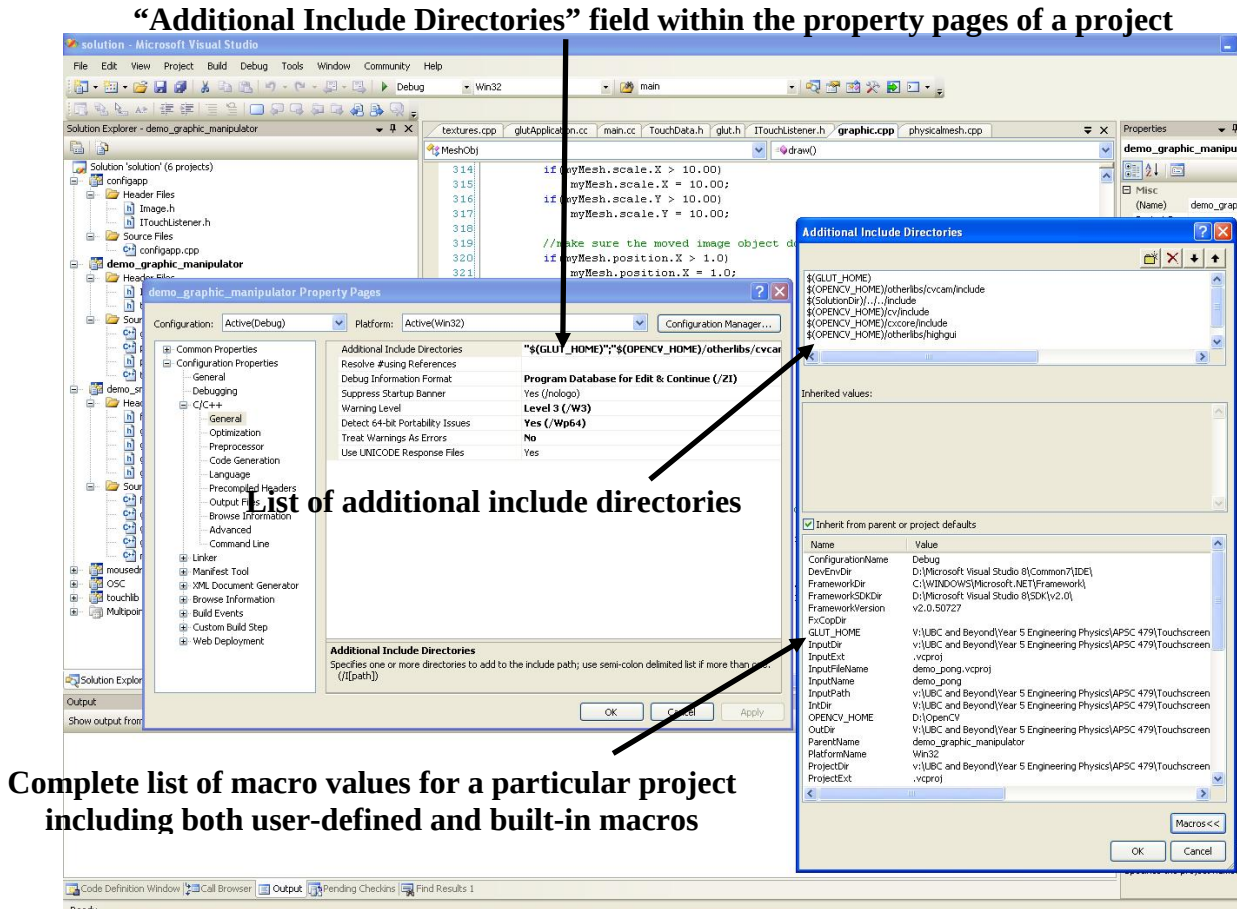


Figure 29. Displaying user-defined macros in Visual Studios.

Each user-defined macro is written in upper-case while the macros in normal case are built-in Visual Studios macros. Built-in Visual Studios macros are defined intuitively but if needed a complete list can easily be found on the Internet. For example, the built-in Visual Studios macro “SolutionDir” resolves to the directory the solution file resides.

Library dependencies are recorded in the project file’s “Additional Library Directories” and “Additional Dependencies” fields accessible by right-clicking on the project within Visual Studios, choosing “Properties” then looking under “Config Properties”→ “Linker” → “General” → “Additional Library Directories”. Macros are also used to resolve absolute library directory location to ease portability as described in the next paragraph. The “Additional Dependencies” field, used for stating filenames

of specific libraries to include, is similarly located under “Config Properties”→ “Linker” → “Input” → “Additional Dependencies”.

User-macros are definable through property sheets in Visual Studios 2005. Property sheets provide flexibility as a central point of control for porting software development environments to other machines. To see the complete list of property sheets linked to the EECE_478_Game_Project solution file, switch from the Visual Studios Solution Explorer view to the Property Manager view, expand a project by clicking the “+” sign beside it, expand the “debug” or “release” hierarchy and double-click “Base”. Within the Base property pages, go to “Common Properties” → “User Macros”. The lists of user-defined macros are displayed.

To modify the user macros, open “base.vsprops” (located in ~ \EECE_478_Game_Project\EECE_478_Game_Project\Property Sheets) with a text editor and follow the syntactical pattern to redefine or add user macros. Note that the EECE_478_Game_Project solution file must be reloaded in Visual Studios for the effects to be observed. The other two property sheets present in ~ \EECE_478_Game_Project\EECE_478_Game_Project\Property Sheets are characteristics reserved for either debug builds only or release builds only. The debug and release build property sheets should never need to be modified since each inherits from the base.vsprops file.

All user macros are defined relative to the location of the EECE_478_Game_Project.sln file so if the relative directory hierarchy is preserved on the local development machine, as uploaded to SourceJammer, our team’s chosen SCM tool, all header and library file dependencies should be resolved without problems. Our directory hierarchy is shown in Figure 30 below.

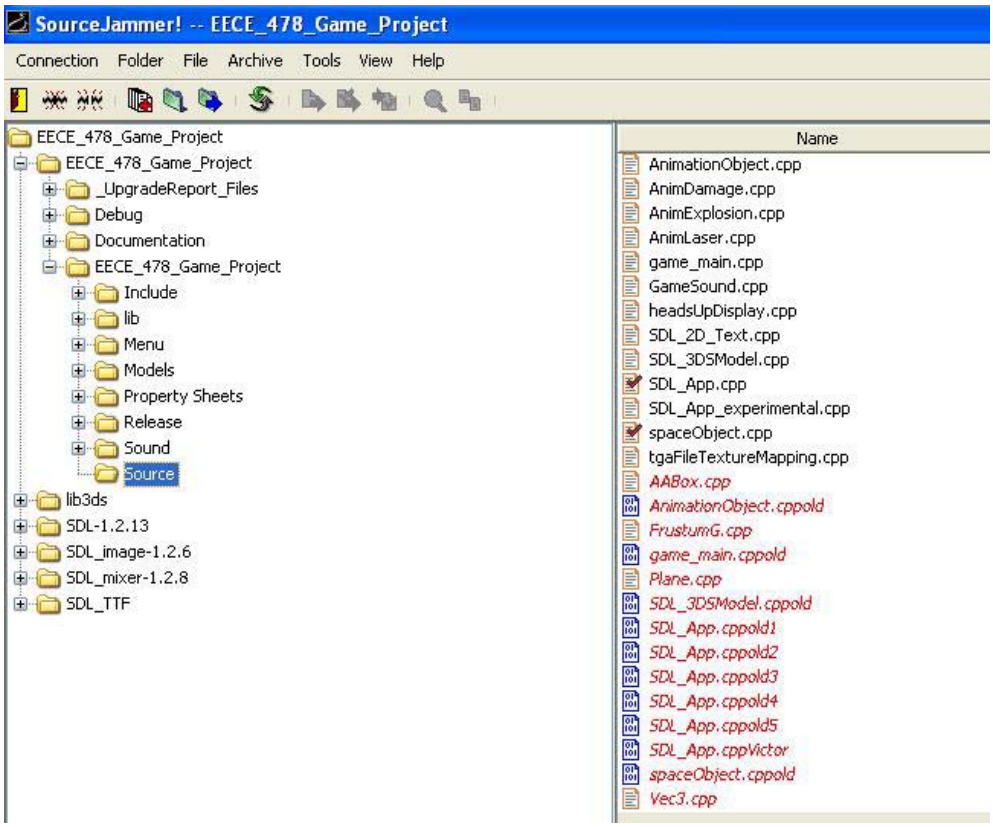


Figure 30. SourceJammer directory hierarchy.

2.16 SourceJammer: An Open-Source SCM Tool

To permit concurrent access to files commonly modified by team members, a software configuration management tool was essential. Our team chose the open-source project known as SourceJammer (www.sourcejammer.org) for its intuitive, simple user interface for checking in and out files while maintaining a file version history. Moreover, SourceJammer has sufficient documentation online and is straight-forward to install, maintain and administer. The server-side of SourceJammer requires an installation of the latest Java SDK as well as the open-source Apache Tomcat application server. However, the client-side merely requires an installation of the current Java runtime libraries and the client application itself.

After each group member's account was created, individual login credentials were distributed to each developer in addition to the Tomcat

application server URL where the SourceJammer server was running. Overall, SourceJammer allowed primary application files to be revised frequently (in some cases more than thirty-five times) without introducing regressions. If regressions did occur, the versioning system permitted previous stable file versions to be recovered instantaneously. Thus, the latest core game files and open-source libraries could be conveniently distributed to each team member and a common working game build was always available. A screenshot of the SourceJammer GUI is shown below in Figure 31.

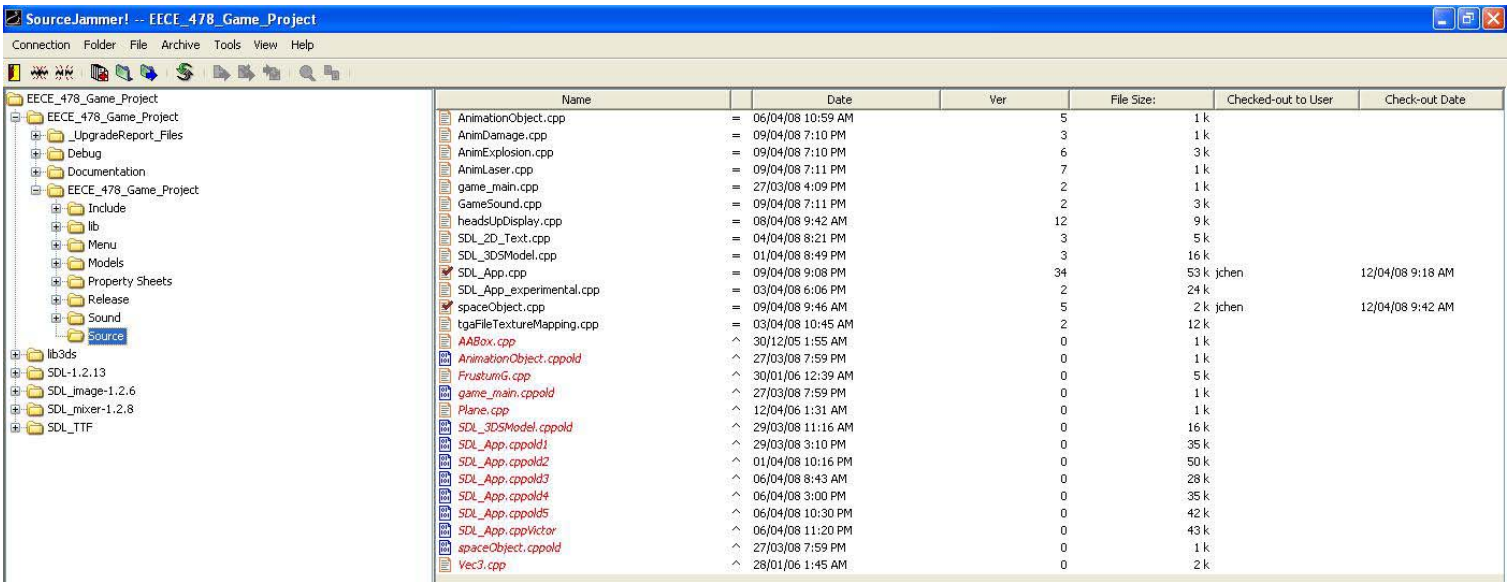


Figure 31. SourceJammer user interface.