
**Summary**
Team project to make a clone of the classic game “Missile Command” without using any of the game code available online.

**Details**
Recreate Missile Command from scratch using only C++, OpenGL, and the most basic OpenGL libraries.

**TeamWork:**
We went with a loosely defined modified Agile approach mixed with aspects of Extreme Programming (you know, especially during the last fews days) this approach had its ups and downs considering that we did not create a generalized style-guide or similar. In the end, I think we all learned a little more about how to work in a peer-team environment.
Github:

C++ in a team environment:
Getting all of the files to work together and build (make) properly on a team project in C++ was interesting. We noticed that apparently even the order that files are added in the Makefile can affect whether or not one file is aware of another’s functions. C++ is a lot more finicky and unforgiving than some other languages.

Communication:
We went with Google Hangouts as our main communication tool to allow us to do minute-by-minute updates even on our phones, as well as give us the ability to do virtual meetings and share our screens. This proved to be very useful because we could ask and answer questions quickly even when not near a computer.
We chose to do weekly team-meetings on Sundays so that we could all attend whether in person or virtually. We also did a bunch of scrum meetings while in class or just impromptu.

Game Aspects:
Game Play
Multi-level game-play with scoring, and increased difficulty at each new level.

Requirements:
1. Tracking of mouse movement in order to aim/shoot missiles at point of click
The player needs to be able to shoot down enemy missiles raining down from the sky by having a friendly missile be shot out to the point where the mouse is currently being pointed to.
2. Structures (Missile silo(s) and cities)
Objects that represent cities must be rendered at the bottom of the screen to portray the structures that the player must protect. These also represent how many lives the player has before it is Game Over. They will be destroyed if an enemy missile collides with one of them.
3. Collision detection of nukes, missiles, and structures
When two objects collide with one another, an event must occur. When a missile hits a city, both must explode. When a friendly missile hits an enemy missile, they both must explode. When a nuke is activated, all missiles must be destroyed.
4. Rendering of missile and nuke trails/paths/explosions
The actual missiles, explosions, and enemy missile trails must all be rendered to clearly view what trajectory they are currently heading. This will help the player see if an enemy missile is currently heading towards a city.
5. Difficulty or level system
A difficulty based level system needs to be put into place to ensure the game is not too easy and will eventually end. As the player progresses in killing enemy missiles, the level will change and the number of enemy missiles spawned will increase accordingly. This creates an internal difficulty system that a player has no choice but to face as they get better at the game.
6. Sound effects
Audio would be used to cue the player that something has occurred in the game such as missiles firing, missiles exploding, cities being destroyed, and the beginning of a new level.

7. Scores/menus

Menus would allow the player to start the game and show them how to play the game through an informative menu image or text. Scores would show the player if they have done well in playing the game. A higher score naturally means a good player. A low score would show to the player to keep on practicing.

**Design:**

Gameplay is straightforward and similar to Missile Command. The goal of the game is to defend your cities from the incoming enemy missiles by shooting them down before they strike.

**Defense Missiles:**
Shoot defensive missiles similar to original game.

**Nuke’em:**
JBC Special key feature to kill all enemy missiles at once.

**Enemy Missiles:**
Generate enemy missiles from the top of the screen which will hit random locations on the floor. Speed and number of missiles will increase with level. Missiles will sometimes branch at higher levels.

**Radar:**
A radar system would be implemented which would only show enemy missile within its radius for Daniel’s special feature. This could be toggled with a key press.

**Smart Missile:**
A special missile would be designed which would attempt to avoid interception by the defense missiles.

**Levels:**
A cap on the number of enemy missiles would be set so that the game knows when each level is over. Once the enemy missiles for the level are used up, the score will be added up and a new level started. The difficulty level will increase as the level increases (based on missile speed and number) but the score multiplier will increase as well.

**Menu:**
A menu will be used in order to navigate the player through the options available in the game. The menu should at least allow the player to start the game, access settings, and exit the game.

**Audio:**
Audio is necessary in order to give auditory cues to the player that something has occurred. This would be explosions, missile launches, music, and simple sound effects for smaller functions such as clicking, alert sounds, and mouse clicks.

**Texture/structures:**
The game needed to be similar to its clone, Missile Command. So the basic textures of missiles, cities, silos, terrain, and background should be there. They would also need to be transparent to blend into the game.

**Civilians:**
Civilians running across the terrain adds to the atmosphere of nuclear war and explosions going off everywhere. Naturally, there would be people running around everywhere if they weren’t already vaporised.

**UFO:**
A UFO is an extra feature that adds to the fun and difficulty of the game. It hovers around the top of the screen and randomly shoots down missiles to the unsuspecting player. It cannot be destroyed.
Coding:

Summary:
We chose to start with the bare bones file from Lab1. The idea was that we would likely learn the most from starting with a very small foundation to force ourselves to learn all of the details of OpenGL instead of just using libraries that take care all of the hard work. It allowed us to get a deeper understanding of OpenGL. Like learning the basics of math before using a calculator, now we know what is going on behind the scenes of OpenGL. We may never want to go without more complex OpenGL libraries in the future, but now we know why!

Menu/Buttons:
In order to allow the player to navigate through the game objects had to be created that would resemble a working menu. The design for the buttons was based off of our work on the Waterfall Model homework assignment. From that example code I was able to draw a set of simple rectangles.

```c
void drawMenu(Game *game)
{
    for (int j = 0; j < BUTTONS; j++) {
        game->buttons[j].x = (j*95);
        game->mButton[j].y = WINDOW_HEIGHT - 200;
        game->mButton[j].width = 120;
        game->mButton[j].height = 25;
        game->mButton[j].center.x = WINDOW_WIDTH / BUTTON_X;
        game->mButton[j].center.y = WINDOW_HEIGHT - game->buttonSpacer[j];
    }
}
```

The next goal was to add detail to the shape such as color and text as well as a hover effect when the mouse entered the boundaries of the shape. The hover effect was a simple modification to the collision detection code that modified a variable tied to an index number that tells the render code which of the two colors to use when rendering the menu shapes.

```c
Shape *s;
if (gameState(game) == 1) {
    for (int j = 0; j < BUTTONS; j++) {
        s = &game->mButton[j];
        if (savex >= s->center.x - (s->width) &&
            savex <= s->center.x + (s->width) &&
            savey >= s->center.y - (s->height) &&
            savey <= s->center.y + (s->height)) {
            game->mouseOnButton[j] = 1;
        } else {
            game->mouseOnButton[j] = 0;
        }
    }
}
```
Audio:
The audio class was based on Gordon’s OpenAL example. That example helped point me in the right direction as to how to go about adding basic audio to our game. In order to keep everything in scope and easily store the data I went with a class.

This class allowed me to easily store and call any audio while keeping everything neat and organized. Next was coding a function to play these audio files while allowing them to overlap which was accomplished with the code below. Max was determined based on the integer passed to the function.

```
class Audio
{
    public:
        //Variables
        ALCdevice *device;
        ALCcontext *context;
        Aluint alsource;
        Aluint alBuffer;
        Alint source_state;
        Int buffer[TOTAL_SOUNDS];
        float gVolume;
        //Constructor & Destructor
        Audio();
        ~Audio();
        //Class Prototypes
        loadAudio();
        void playAudio(int num);
};
```

Level-to-Level:
Creating a level-to-level function required a bit of logical thinking. Simply put, the function would set a timer, calculate your bonus points, check if the game over conditions were met, otherwise continue on and then take you to the next level. Timers were used in order to determine how long you have been and should be in the function. Every loop through this function would store the important variables in a struct called levelInfo.

This allowed the rendering of the bonus score function to correctly output which was a simple render function that took in the arguments passed in from the endLevel(...) function.

```
if (clockReset) {
    //printf("Clock Reset\n");
    time(&start);
    timer = 0.0;
    clockReset = false;
}
```

```
if (rCount <= rMissiles || cCount <= rCities || game->gState == 99) {
    return;
}
```

```
else if (rCities == 0) {
    game->gState = 99;
    resetLevelEnd(game);
    return;
}
else if (cCount != cCities) {
    //Reset Game State once delay is reached
    if (difftime(end, start) >= delay) {
        printf("Level: %d\n", game->level);
        resetLevelEnd(game);
        game->gState = 0;
    }
else {
    //Store calculated data
```
Texture/structures:

I used Gordon’s PPM files to transform regular jpg or png images to transparent textures that could be binded onto different rendered shapes. The trick I needed to figure out was the alpha data image function that gave a regular rgb image an alpha value. I then enabled GL_BLEND and activated the BlendFunc with alpha parameters to give all my textures a silhouette and transparent look to them. I also saw that I needed to use BindTexture with parameter 0 in order to get the individual textures from interfering with the other textures. The TexCoord2f I learned were used to show what position the texture would be shown. I messed with the values a little bit until I got the right combination for the texture to show its front to the screen.

Civilians:

I used lab1.cpp’s particle function to create numerous civilians that could move across the screen. I made it so they spawn based on what x and y values are passed in as parameters. So they can spawn from buildings or explosions if we wanted. Is they are touching the floor, velocity would carry them to the right to simulate running from the nuclear holocaust.
void ufoPhysics(Game *game) {
    if (game->ufoOn == 0)
        return;
    UFO *u = &game->ufo;
    // UFO *
    // move ufo...
    int addgrav = 1;
    // Update position
    u->pos.x += u->vel.x;
    u->pos.y += u->vel.y;
    // Check for collision with window edges
    if ((u->pos.x < -160.0 && u->vel.x < 0.0) ||
        (u->pos.x >= (float)WINDOW_WIDTH + 160.0 && u->vel.x > 0.0)) {
        u->vel.x = -u->vel.x;
        addgrav = 0;
    }
    if ((u->pos.y < -470.0 && u->vel.y < 0.0) ||
        (u->pos.y >= (float)WINDOW_HEIGHT + 470.0 && u->vel.y > 0.0)) {
        u->vel.y = -u->vel.y;
        addgrav = 0;
    }
    // Gravity
    if (addgrav)
        u->vel.y -= 0.05;
}

**UFO:**

The UFO is made by having a key pressed to on or off. It’s rendered near the top and the position will follow the velocity. It moves in the opposite direction when it hits the edges of the window. The gravity boolean is turned off and on based on the position that it is in to imitate hovering.

**Technical challenges:**

**Defense Missiles (JBC):**

Getting the math right was interesting as well as making sure that the missiles stopped exactly at the mouse-pick coordinates because the missile was jumping up to 100 pixels in each iteration of the rendering function. Daniel gave me a hint that helped lead to the solution.

**Audio (JR):**

Figuring out how to initialize and store the required OpenAL functions was a difficulty I faced. Initially, I had initialized the required variables such as devices, context, sources, and buffers in ‘int main()’ however I was unable to easily call and pass variables without running into a wall of ‘Out of Scope’ errors. I was able to remedy this by creating a class called Audio in which I declared and initialized the required variables. This allowed me to quickly and easily add new sounds and continue on to figuring out my next challenge which was how to store and play several sounds at once.

This part was a bit easier to figure out as I had already done the hard part by creating a class for my audio. At this point I could simply create any sized array of sources (however many I needed) and store a specific buffer in each of these sources. Now, how was I gonna call these sources without calling the same one?
While not user friendly, I was able to allow the call of multiple sources, without creating a new one every time I needed it, by creating several sources for commonly used sounds. These included the missile explosions, missile collision explosions, and missile fired sound effects. The playAudio function would simply iterate to a max of 9 from its original index number until it found a source that was NOT playing and would then use that source to play the effect. Although, if all sources were in use the function would return and print out a statement in the terminal stating that all sounds are currently in use. The max of 10 sources per sound effect was based on my judgement since no more than 10 missiles are usually on screen. In the end, I managed to create the desired effect and sound effects worked as I had initially intended them to.

Textures(JG):
Figuring out how an image is turned into a transparent cutout of itself was very difficult. All the code was there in the PPM file but was not very clear at all. Me and Jose Reyes teamed up to figure it out. Took us about a good 3 days to see that the silhouette data made in PPM.cpp was the texture we needed. Now all I had to do was make images with black backgrounds to generate all the textures we needed for our game. I also figured out on my own that I needed to set the bind texture function back to 0 in order to keep the texture that was rendered from messing up all the other colors and textures.

Enemy Missiles(DT):
Early on it took some time to figure out the random aspect of the missiles but still making them land inside the screen. I would spawn the missiles at a random location and then give them a random x and y velocity to give different angles. But many would end outside the edges of the screen. After trying lots of different methods, I finally just precoded a random start and destination for each missile and calculated the necessary velocity based on that. Also for the radar it took some time figuring out how to get the look I wanted. I started with just an expanding circle and checking for the missiles within the circle. But I ended up with a set circle radius and then animated triangle fans inside the circle. The physics of the radar slowly changed the transparency of the triangle fans to give the look of fading radar.

Debugging/Refactoring/Optimization:
Some of us used Unit testing on some of our functions to ensure correctness reliability and performance. For instance in the locations of explosions to ensure that they could be placed anywhere on the screen without conflict.