MSIM 742

Synthetic Environments

Assignment Two

Mark Brosche

MSVE Master of Engineering Student

mbros006@odu.edu

(407)-325-2818

Distance Graduate - Outside Virginia

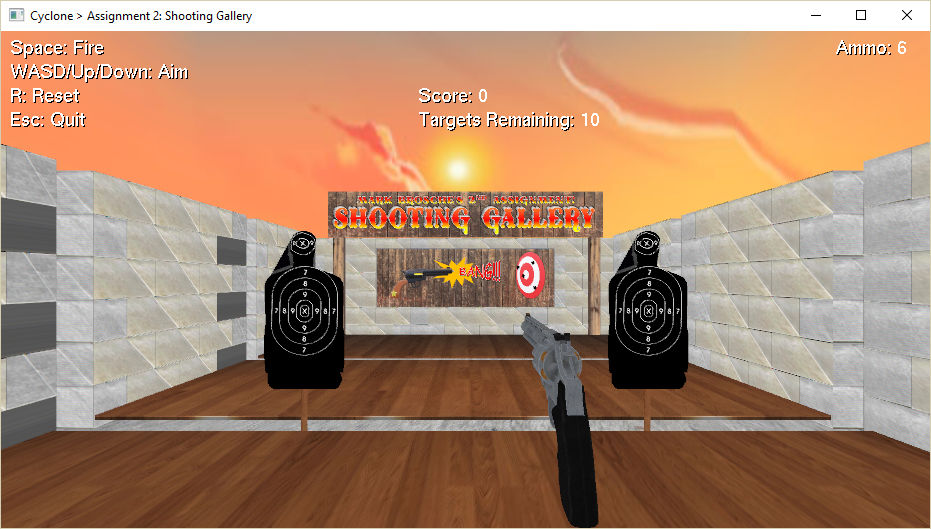
March 4, 2018

## Introduction

For this assignment, the goal was to produce an FPS style shooting game with the Cyclone engine and based on the “BigBallistic” demo. The game itself should fashioned such that a player can aim around and shoot targets in different locations. As the player successfully shoots the targets they should fall using the cyclone rigid body physics and disappear and the score should increase until there are no targets remaining. The learning objectives were to gain a working knowledge of development in the Cyclone engine and focusing on collision detection and particle physics. As the Cyclone engine is based in OpenGL/C++, reinforcement of knowledge from previous classes covering OpenGL and object-oriented programming (MSIM 541) is expected.

A video of the application in action can be found [here](MSIM-742_A-2_Brosche-Mark_Video.mp4).

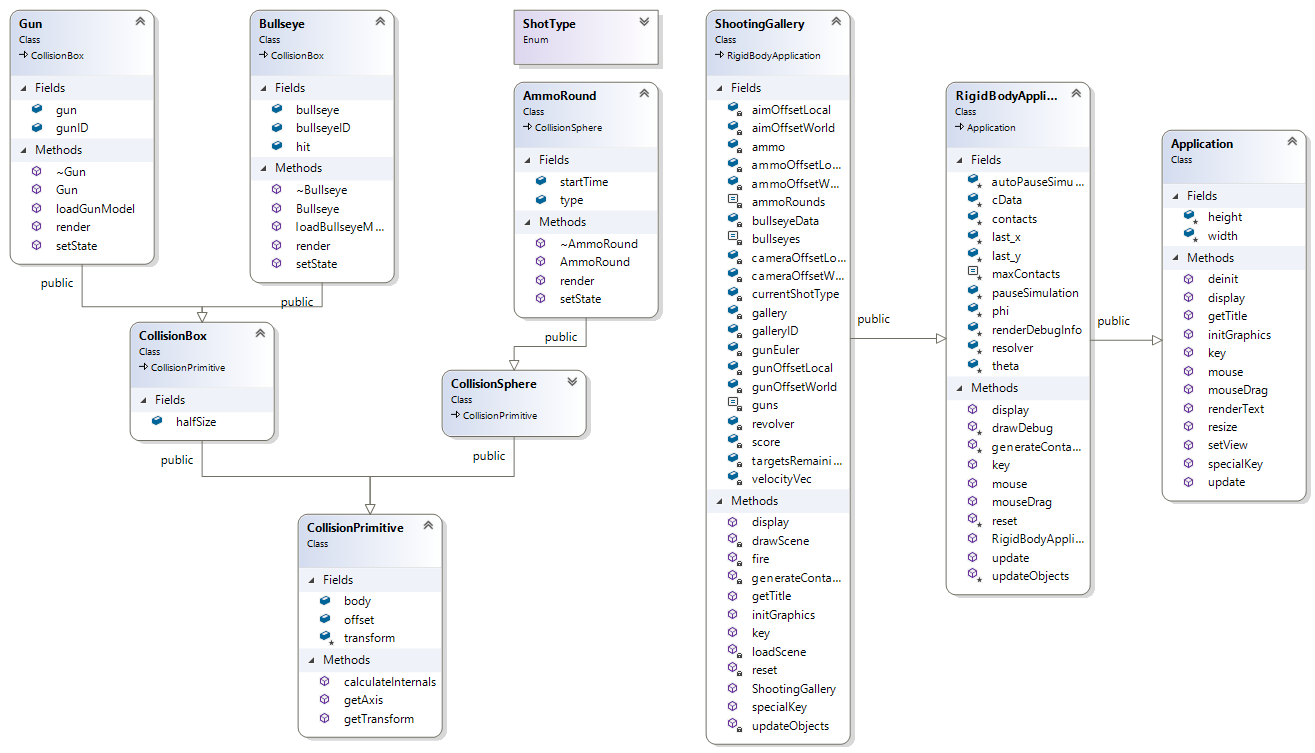
The application executable can be run [here](Debug/GamePhysicsApp.exe.lnk).



**Figure 1. Application Starting Point**

## Program Design

This section identifies the files, functions, models and variables used in the project. Also contained are instructions for running and controlling the application. Since this project was built using the “BigBallistic” demo, only the new additions and changes will be discussed. 3D models were imported using the OBJ parser provided in MSIM 541 with classes and methods outlined below. Another inclusion from MSIM 541 is the Utility.cpp which contains methods for computing rotation matrices and printing strings.



**Figure 2. The class hierarchy of the Shooting Gallery Application.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Table 1: Source Files (BigBallistic files listed in blue): | | | | |
| Source | Headers | Models | Textures | Fonts |
| ObjModel.cpp  PPMImage.cpp  Utility.cpp  ShootingGallery.cpp  App.cpp  Body.cpp  Collide\_course.cpp  Collide\_fine.cpp  Contacts.cpp  Core.cpp  Fgen.cpp  GamePhysicsApp.cpp  Joints.cpp  Particle.cpp  pcontacts.cpp  pfgen.cpp  plinks.cpp  pworld.cpp  random.cpp  stdafx.cpp  timing.cpp  world.cpp | BoundingBox.h  Face.h  Material.h  Normal.h  ObjModel.h  PPMImage.h  Texture.h  Utility.h  Vector3D.h  Vertex.h  App.h  Body.h  Collide\_coarse.h  Collide\_fine.h  Contacts.h  Core.h  Cyclone.h  Fgen.h  Joints.h  OGL\_header.h  Particle.h  Pcontacts.h  Pfgen.h  Plinks.h  Precision.h  Pworld.h  Random.h  Stdafx.h  Targetver.h  Timing.h  World.h | gallery.obj  revolver.obj  target.obj | Bang.ppm  Concrete\_floor.ppm  Diffuse.ppm  Left-sign.ppm  Light-Texture-Brown-Wood-HD-Wallpaper1.ppm  Oldwood.ppm  Right-sign.ppm  Sign.ppm  Sky\_sunset.ppm  Stone\_Masonry\_Multi.ppm  Stone\_Masonry\_Multi1.ppm  Stone\_Masonry\_Multi2.ppm  Stone\_Masonry\_Multi3.ppm  Stone\_Masonry\_Multi4.ppm  Stone\_Masonry\_Multi5.ppm  Stone\_Masonry\_Multi6.ppm  Stone\_Masonry\_Multi7.ppm  Target2.ppm | Coney Island.ttf |

### Key Variables & Functions:

|  |  |
| --- | --- |
| Class: ShootingGallery - The main demo class definition. | |
| Variables:  Public:   * ammoRounds: const static unsigned * ammoCount: int * ammo[ammoRounds]: AmmoRound * guns: const static unsigned * revolver[guns]: Gun * bullseyes: const static unsigned * bullseyeData[bullseyes]: Bullseye * currentShotType: ShotType * score: int * targetsRemaining: int * galleryID: int * gallery: ObjModel * cameraOffsetLocal: cyclone∷Vector3 * cameraOffsetWorld: cyclone∷Vector3 * aimOffsetLocal: cyclone∷Vector3 * aimOffsetWorld: cyclone∷Vector3 * gunOffsetLocal: cyclone∷Vector3 * gunOffsetWorld: cyclone∷Vector3 * ammoOffsetLocal: cyclone∷Vector3 * ammoOffsetWorld: cyclone∷Vector3 * gunEuler: cyclone∷Vector3 | **Description:**   * Holds the maximum number of rounds that can be fired. * Mutable count of ammunition remaining in the weapon. * Holds the particle data. * Holds the number of guns in the simulation. * Holds the gun data. * Holds the number of bullseye targets in the simulation. * Holds the bullseye data. * Holds the current shot type. * Records the number of targets hit. * Store the OBJ file for the static scenery. * Hold offset vectors of Camera, aim, gun, and Ammo from the origin in Local and World-space. |
| Methods:  Public:   * ShootingGallery(): ShootingGallery * loadScene(): void * drawScene(): void * initGraphics(): void * reset(): virtual void * getTitle() * fire(): void * updateObjects(cyclone::real duration): virtual void * display(): void * generateContects(): virtual void * key(unsigned char key): void * specialKey(int specialKey): void * getApplication(): Application | **Description:**   * Creates a new demo object. * Read in and create a call list for the static scenery. * Draw the static scenery. * Sets up the rendering. * Resets the position of all the boxes and primes the explosion. * Returns the window title for the demo. * Dispatches a round. * Processes the objects in the simulation forward in time. * Display world, objects, render text and update the camera view. * Build the contacts for the current situation. * Handle a keypress. * Handle a special keypress (Arrow keys for this app). * Called by the common demo framework to create an application \* object (with new) and return a pointer. |
| Class: AmmoRound - The AmmoRound class stores the information for instantiating and updating bullets, physics is applied when bullets are fired from the gun. | |
| Variables:  Public:   * type: ShotType * startTime: unsigned * velocityVecLocal: cyclone∷Vector3 * veloctiyVecWorld: cyclone∷Vector3 | **Description:**   * holds the type of the shot (pistol or UNUSED) * holds the time of instantiation for a shot |
| Methods:  Public:   * AmmoRound() * ~AmmoRound() * render(cyclone::Vector3 gunEulerAngle, cyclone::Vector3 ammoCamOffset) : void * setState(ShotType shotType, cyclone::Vector3 position, cyclone::Vector3 angle): void | **Description:**   * Constructor * Destructor * Draws the shot. * Sets the shot to a specific location and velocity. |
| Class: Bullseye - The Bullseye class stores the information for instantiating and updating targets, physics is applied when collisions with bullets are detected. | |
| Variables:  Public:   * Bullseye: ObjModel * bullseyeID: GLuint * hit: bool | **Description:**   * holds a 3D OBJ file * OBJ reference for glCallList() * Holds the hit status of a bullseye. |
| Methods:  Public:   * Bullseye() * ~Bullseye() * loadBullseyeModel(): void * render(): void * setState(cyclone::real x, cyclone::real z): void | **Description:**   * Constructor * Destructor * Reads in .obj file and creates a call list for it. * Draws the bullseye. * Sets the physics properties of the bullseye. |
| Class: Gun - The Gun class stores the information for instantiating and updating a Gun model, physics is not applied. | |
| Variables:  Public:   * gun: ObjModel * gunID: GLuint | **Description:**   * holds a 3D OBJ file * OBJ reference for glCallList() |
| Methods:  Public:   * Gun() * ~Gun() * loadGunModel() * render(cyclone::Vector3 gunEulerAngle, cyclone::Vector3 gunCamOffset): void * setState(cyclone::Vector3 position): void | **Description:**   * Constructor * Destructor * Reads in .obj file and creates a call list for it. * Draws the gun. * Sets the physics properties of the gun. |
| ObjModel.cpp (OBJ Parser) | |
|  |  |
| Public:   * Readfile(string): void   Protected:   * DrawMaterials(string, bool): void | **Description:**   * Updated to read and generate textures from PPM files. * Updated to bind textures to OBJ meshes. |
| PPMImage.cpp (for OBJs with textures) | |
| Public:   * width: int * height: int * image: unsigned char\*   Private:   * c: char * buf[128]: char * file: ifstream * outFile: ofstream | **Description:**   * texture image width * texture image height * pointer char array to image pixels * variable for parsing ascii text * array for * variable for accessing files for input into memory * variable for writing files |
| Public:   * PPMImage(): void * ~PPMImage(): void * AllocateMemory(int, int): void * ReadFile(string): void * VerticalFlip(): void * WriteFile(string, string): void   Private:   * CheckComment(): void | **Description:**   * Constructor * Destructor * Creates a new empty array of the size of the image to be recorded to it. * Reads image file into memory at char \* image * Iterates through the image array to vertically flip it. * Writes the image array to file in either P3 or P6 PPM format. |
| Utility.cpp – used for Rotations and printing strings | |
|  |  |
| Public:   * printString(string str): void * printLargeString(string str): void * computeRotatedVector(cyclone::Vector3 input, cyclone::Vector3 angle): cyclone∷Vector3 | **Description:**   * Prints a string to the OpenGL window. * Prints a larger font size string to the OpenGL window. * Computes a rotation matrix for a given position vector and Euler vector. |

### Instructions to run code:

Run **“GamePhysicsApp.exe”** in the **/Debug** folder

Inputs are as follows:

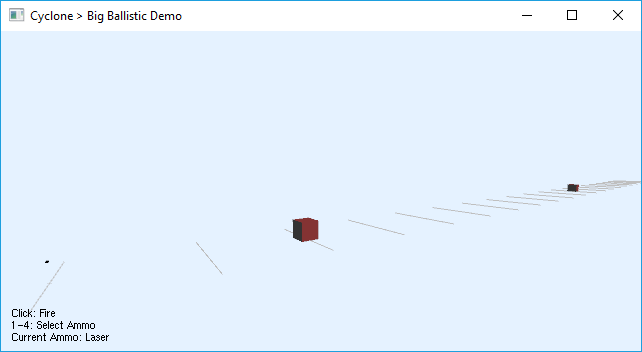
* UP – translate up in the positive y direction.
* DOWN – translate down in the negative y direction.
* w/W – look up.
* s/S – look down.
* a/A – turn left.
* d/D – turn right.
* Space – fire a bullet.
* R – Reset (resets gun and targets and applicable variables to their starting states.
* ESC – quit application.

## Results

I have organized my results with this project with respect to the tasks section of the assignment document.

### 3.1 Studying the Base Project

Looking at the Big Ballistic Demo, we can tell that it is a derived class of the “RigidBodyApplication” class and it contains classes for the box objects and the ammunition to calculate the results of collisions of the respective rigid bodies of these glut primitives. The view is statically set in the display method and the input options are printed to the bottom left corner of the window. Depending on the ammo selected, a round will instantiate with a pre-set velocity from the point on the left of the view and head toward the two boxes. On collision with a round, the boxes will move based on the mass, velocity, and point of impact of the round on the box. Objects in the scene are updated at a rate flexibly defined in the Application Class as being between 0 and 0.05 depending on how fast the application/PC can compute a frame.



**Figure 3. Big Ballistic Demo.**

### 3.2 Building my Game Scene

*Files:* [*ShootingGallery.cpp*](GamePhysicsApp/ShootingGallery.cpp)*,* [*gallery.obj*](GamePhysicsApp/Models/gallery.obj)*,* [*revolver.obj*](GamePhysicsApp/Models/revolver.obj)*,* [*target.obj*](GamePhysicsApp/Models/target.obj)

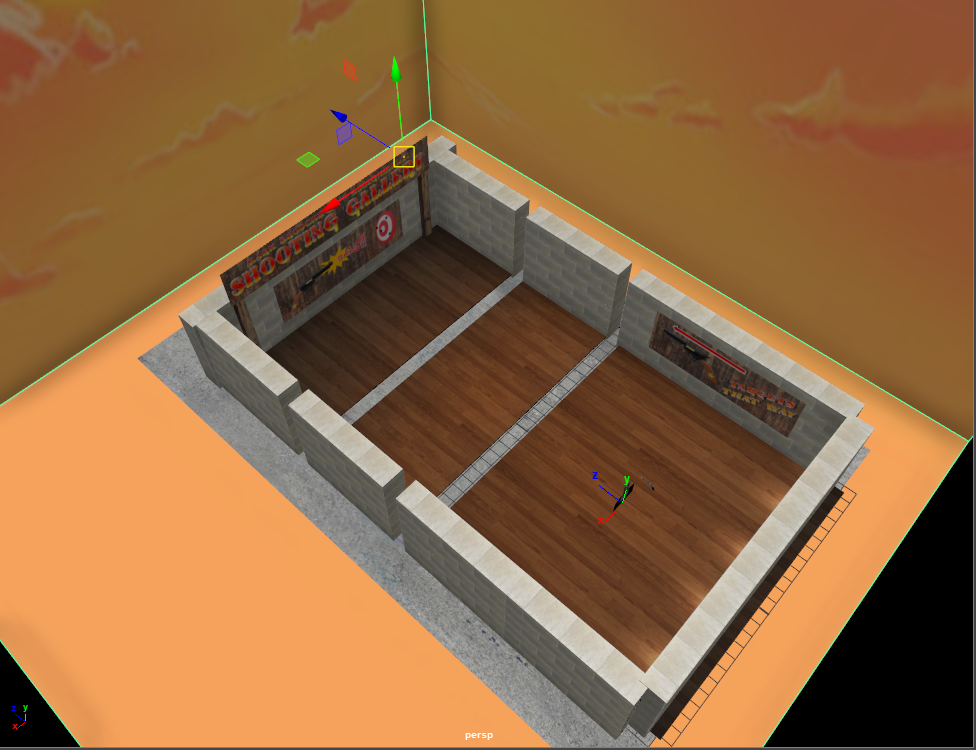
To build my game, I started with the Big Ballistic demo project, and modified and added to it heavily.

I chose to fashion my game after a wild west-themed shooting gallery. The scene was created in Maya using primitive planes and cubes as shown in the figure below. Maya was specifically helpful in ensuring the sourced models and the scene were reasonably of the same scale. The gallery consists of two tracks for targets to move along, wood panel floors, cinderblock walls and sign textures that I created myself in PowerPoint. The scene also contains a skybox cube map I fit onto an inverted cube primitive. These elements were exported from Autodesk Maya to the OBJ format and brought into the game using the OBJ and PPM Image parser used in MSIM 541.

I created a method in the ShootingGallery class to load the scene (loadScene()) within the initGraphics() function, and then to redraw the scene (drawScene()) using its glCallList() every time display() is called.

The revolver and target models were sourced from cgtrader.com and free3D.com respectively. The target model I made further additions and modifications to by flattening and adding a support stick to the bottom.

As the scene loads, 10 targets are populated to run from end to end on the two tracks (5 front and 5 back) at a constant velocity. Targets do not collide but will phase through each other.



**Figure 4. Gallery scene in Maya (left) and target and gun models in Maya (right)**

Instructions for playing the game can be found at the end of Section 2 of this report.

\*Aiming is limited to ±70° about the x axis and ±90° about the y axis.

### 3.3 Gameplay

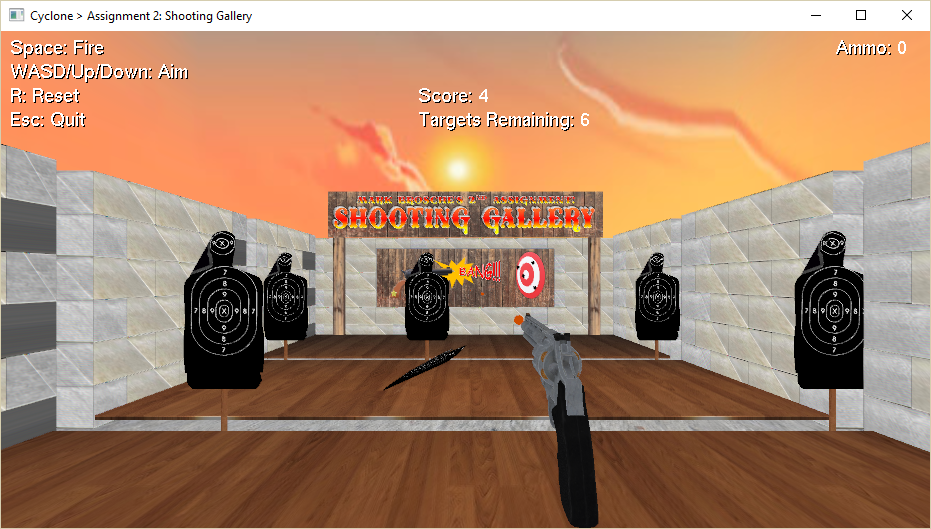
The Bullseye class targets are a modified version of the box class used in the Big ballistic demo. The bullseye class is different from the box class in that it uses a custom 3D model and moves at a constant velocity on a predefined path, but it does generate collisions with the ground plane and with ammo rounds where it will apply physical changes to the object transform.

The AmmoRound class is for the most part unchanged from Big Ballistic, using the Pistol case and with a velocity vector variable added for computing the adjusted instantiation point when the player changes the aim of the gun.

The Gun class was also a copy of the box class of the original Big Ballistic demo, with the exception of the loadGunModel(). The gun transform is recomputed in the render() method as it is the only one called on every update. The gun is not centered in view but is offset to the side and is stationary with respect to the camera. This means that the gun’s transform must be updated based on this offset, as well as the change in angle of the camera, which in this case is done via glRotate and glTranslate in a specific order so that the gun effectively orbits the player. The Rigidbody∷set[…] methods were not used because they rely on world space coordinates and default to the center of the model. When any of the directional keys are pressed, the offsets for the aim and ammo are recomputed with a rotation matrix and the scene updates on the next call of the display function.

The upDateObjects() method moves bullseyes at a constant velocity from one end of their track, to the other. They can only collide with bullets and will behave in a breakaway manner. On collision with a bullet, a bullseye’s velocity is reset to zero, gravitational acceleration is added, and a force at the point of collision is created to give the target a realistic reaction to being shot at the specific point of collision; all methods that are available in the Rigidbody class.

The ground collision plane has been set further along the negative y-axis to give the targets room to fall before being scaled to zero size and put to sleep. Targets are dealt with this way as opposed to being set as the ‘UNUSED’ enumeration like in the AmmoRound class because they do not become available for use again until the game is reset.



**Remaining ammo before reloading is required.**

**Bullet**

**Figure 5. Bullets firing from the revolver knocking down a bullseye.**

Since I chose a revolver as my gun, I decided to allow only 6 shots to be fired in quick succession and then reloading, and to show the player what is happening I read out the ammo count on the top right of the screen.

When a player successfully knocks down all the bullseyes, a message appears saying “You Win!”.



**Figure 6. Win text appears below Score in the Game window.**

Lastly, if the player aims in an unsafe angle, such as toward other potential players or the sky, a warning message will pop up to encourage safer behavior.



**Figure 7. A warning appears if you aim at an unsafe area of the scene.**

## Conclusion and Discussion

I had a great deal of fun with this project. It challenged me in many ways as I will discuss below, and I was able to find creative ways to overcome or power through most of those challenges.

Accomplishments:

* Having the gun offset from the center of view was a challenge that I’m happy I succeeded with. It would have been easier to place the gun and shot starting point minimally offset from the origin but going for a “shooting from the hip” style forced me to understand the way rigid body transforms in Cyclone behaved vice glTanslate and glRotate, and computation of rotation matrices.
* I also chose to move the entire camera when the gun moves which is how a typical first-person-shooter works. This was also a challenge to get the camera and gun moving and rotating the same amount for a given key press. Ultimately, the Rigidbody methods for movement were not useful despite being more smoothly animated and rotation matrices were required.
* Implementation of Rigidbody physics for this project was actually one of the least challenging aspects for me.

Difficulties:

* Getting a handle on the rotations and the order in which to compute them was difficult. I had to experiment with the big ballistic demo quite a lot to really get a good understanding of the order of operations for transformations (e.g. translate-rotate-scale) and the difference in effect between using glTranslate and glRotate in the render() methods vice setPosition and setOrientation in the setState() methods. I believe my implementation was ultimately simpler because I took the time to empirically figure out what worked best.
* I also had trouble with lighting and shading in this project initially, which led me to put textures on everything and avoid the problem altogether. I was able to discern later that GL\_TEXTURE\_2D was not getting disabled when needed and that prevented other colors from being expressed. Still, the result is that my objects look a little ‘dead’ because they are unlit.

Possible Improvements:

* Smoother motion for aiming around the scene. This problem could be stemming from framerate and hardware capabilities, or I might not be utilizing the cyclone engine as it is meant to be and it resists my attempts to move the camera and gun kinematically as opposed to with physics. I believe if there is a way to synchronize or equate angular velocity with linear velocity via the keyboard controls, then that might solve it. Googling this topic also brought up the idea that gluLookatAt() is perhaps not the best way to enable an FPS game since one is not usually looking at only one spot.
* Colliders for the scenery. Bullets and targets do not collide with the scenery currently, but it would be more realistic if they did. This would require refactoring my scene elements to import modularly as opposed to one aggregate model.
* Animations for recoil and reloading of the gun. Because the gun has a Rigidbody, recoil from each shot should be possible, but would rely on using timers or something similar to coroutines in Unity to bring back to a “ready to fire” state. Reloading would also require separating out the pieces of the aggregate gun model into the revolver, latch, hammer, shells, and casings. Not a trivial amount of work!
* Models for the bullet. This could be simple, but would also require a better understanding of how to set orientation with the Rigidbody so the bullet local z points in the same direction as the gun local z.
* Lighting and shading. As my models are textured and unlit, it would look better to have them cast shadows and exhibit more material properties.