3D Aim Trainer

Caldara Davide - 973885
Valota Andrea - 982544

September 2021

1 Introduction

The goal of this project is to create an interactive aim trainer. In this kind of application the player has to aim and shoot to a set of static or moving targets in order to improve his skills with mouse and keyboard. The player usually finds himself in a room and is required to face different tasks. The main focus of this project is to show different graphical effects by means of OpenGL[4] low-level functions. In this project we used *OpenGL version 4.1* and C++ language. The use of different effects, materials and target layouts in each room allowed us to implement three unique game modes, also customizing vertex and fragment shaders. The language adopted by the shaders is GLSL[3] version 410. Other techniques like normal mapping, shadow mapping and skyboxes are present in all the scenes. We used the Bullet 2.83 library[1] to manage the physics of our application and implemented a ray casting function to make collisions between bullets and targets precise and responsive. For HUD rendering we used the FreeType 2.10.0 library[2].

2 Game modes

In our application we have created three different game modes which take place in three different rooms. The scene presents the same basic elements in each room:

- three walls that contain the area with the targets
- the targets themselves which are little spheres
- three buttons used to switch between game modes
- two buttons used to change the camera sensitivity
- a skybox
- a directional light used to light up the scene

For each room we used different textures and materials and we implemented shadow mapping and normal mapping to create different lighting effects. In the first room the targets randomly appears within a predefined grid swinging left and right following a regular pattern. Shooting the target will make it move to

another position. In the second room we created a static grid of targets. The one the player has to hit is lighten up in red, while the others are dark. Hitting the right sphere will increase the score and turn on another one. In the third room the targets are bouncing in the area in front of the player. When the player hits a target, the next one will spawn in a random position and a random impulse will be applied to it.

3 First Room: Swinging targets

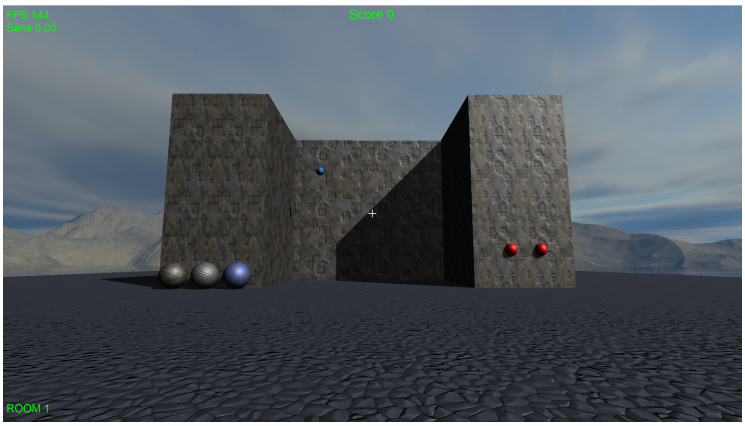

Figure 1: First room

As soon as the player starts the application he finds himself into the first room. He can see a big concrete wall delimiting the shooting area and a stony floor. He can walk around, turn in every direction, but flight is not allowed. On the left the player can find three big buttons that he can use to switch between rooms, each one textured as a reference to the respective room materials. On the right two smaller red buttons can be used to modify the camera sensitivity. The interaction with the buttons is detected through our ray-sphere intersection function. In the targets location the player sees a sphere swinging left to right.

```
1     btTransform t = body -> getCenterOfMassTransform();
2
3     if(deltaTime<1.2f)
4         t.setOrigin(t.getOrigin()+btVector3(0.8 * deltaTime * k,0.0
      f,0.0f));
5
6     body -> setCenterOfMassTransform(t);
```

The code above is responsible of the target motion. The movement depends on *deltaTime* * *k* where *deltaTime* is the time between current and previous frame and *k* is a variable that can be -1 or 1 and dictates the direction of the movement. *K* swaps value every second.
Only one target at a time is present and hitting it will make another one spawn in a new random location.

```
1 if(hit){
2     target_pos = glm::vec3((num_side - (rand()%7)*0.5f)-5.5f, (
      num_side - (rand()%7)*0.5f)-2.5f, 0.7f);
```

```
3       reset_single_target(active_room,target_pos);
4       hit=false;
5   }
6
7   void reset_single_target(GLint next_room_index, glm::vec3 new_pos){
8       btCollisionObject* obj = bulletSimulation.dynamicsWorld->
        getCollisionObjectArray()[10];
9       btRigidBody* body = btRigidBody::upcast(obj);
10
11      btTransform t = body -> getCenterOfMassTransform();
12      t.setOrigin(btVector3(new_pos.x,new_pos.y,new_pos.z));
13      body -> setCenterOfMassTransform(t);
14
15      increaseScore();
16  }
```

When the player hits the active target we generate a new spawn point for the next one (line 2) and we update the rigidBody position associated with the target with the new one (line 7-15).

3.1 Lighting computation: Directional light

In the first room we only have a directional light illuminating the scene. We compute the color of each fragment as the sum of ambient, diffuse and specular term.

$$fragment\_color = ambient + diffuse + specular \tag{1}$$

The diffuse term is computed as the Lambertian term of the GGX illumination model.

$$lambert = (k_d * surface\_color)/\pi \tag{2}$$

For all rooms we considered $k_d = 3$.
The specular term is computed using the FDG equation.

$$specular = \frac{F(v,h)D(h)G_2(n,l,v)}{4(n \cdot v)(n \cdot l)} \tag{3}$$

The respective components are:

$$F(v,h) = R_f(0°) + (1 - R_f(0°))(1 - (v \cdot h))^5 \tag{4}$$

$$D(h) = \frac{\alpha^2}{\pi((n \cdot h)^2(\alpha^2 - 1) + 1)^2} \tag{5}$$

$$G_2(n,l,v) = G_1(n,v) * G_1(n,l) \tag{6}$$

$$G_1(n,v) = \frac{(n \cdot v)}{(n \cdot v)(1 - k) + k} \tag{7}$$

In this room we used $\alpha = 0.4$ and $R_f(0°) = 0.9$ to simulate reflections made by rough materials like concrete and stone. We customized the fragment shader as follows.

```
1  vec3 finalColor = (1.0 - (shadow*0.7)) * (lambert + specular) *
       NdotL;
2
3  if(NdotL==0.0){
4      finalColor = 0.1*(lambert);
5  }
6
7  if(isTarget){
8      finalColor = (lambert + specular) * NdotL;
9  }
10
11 colorFrag = vec4(finalColor, 1.0);
```

To create shadows we apply to the fragment color a custom coefficient that modifies the final color and darken the shaded areas (line 1). We compute shaded areas using Shadow mapping described in the *Shadow mapping* subsection. We do not apply this coefficient to the target spheres to make them clearly visible in every position (line 7-8).
Also, instead of adding a general ambient component to the whole scene we decided to slightly light up the shaded area adding a mitigated lambertian component of the corresponding material (line 3-4). In this way we avoided abrupt change of lighting. The final fragment color is *colorFrag* (line 11) that takes in input a different value of *finalColor* depending on the case we are considering.

3.2 Physics

To manage physics in our project we used the Bullet library[1]. This library provide specific structures for bounding volumes and compute the physical simulation for each step of time Δt. In our application the maximum value set for this parameter is $\Delta t = 1/144$ which guarantees at least one step of the physical simulation for each frame (considering a maximum framerate of 144 FPS). To manage collisions between objects, we used the Sphere shape provided by the library for targets and buttons and the Box shape for walls and the floor. Using this system we manage the movement of the dynamic targets: in the first room we modify directly the position of the spheres to have a more precise and uniform behaviour, in the third room we apply an impulse in a random direction and let the sphere freely move according to the physical simulation.

3.3 Collisions

To compute collisions between the player's bullets and targets and buttons we implemented a ray-sphere intersection function.

```
1  bool hit_sphere(const glm::vec3& center, float radius, const glm::
       vec3& origin){
2      glm::vec3 oc = origin - center;
3      float a = dot(camera.Front, camera.Front);
4      float b = 2.0 * dot(oc, camera.Front);
5      float c = dot(oc,oc) - radius*radius;
6      float discriminant = b*b - 4*a*c;
7      return (discriminant>0);
8  }
```

The function takes in input center and radius of the collision sphere associated to the target or button and the camera position. It looks for possible intersection between the sphere and the shooted ray. Since we do not need to find the precise

intersection points, we simply return a boolean to notify the intersection.
We decided not to use the physics library collisions for this case because, to shoot a projectile in a straight line, we had to use a sphere with very small mass and apply a very strong impulse. Doing this, collisions resulted very inconsistent since a small sphere going at high speed could go through walls or even targets depending on small differences in the player position. Even changing the parameters of the physical simulation did not solve the issue. For $\Delta t < \frac{1}{1000}$ collisions were still not detected correctly everytime despite the drop of framerate. Increasing the number of substeps of the simulation to 50 didn't work as well. To simulate the projectile we still shoot a small sphere so that the player has a visual feedback whenever he fires, but it is only rendered and has no function in the collision system.

3.4 Shadow mapping

Shadow mapping is a technique used to create shadows in the scene. To do this we need to perform two render passes. In the first pass we render the scene from the point of view of the light. Everything we see from this point of view is lit and everything else is in shadow. So we render the scene in this way and store the resulting depth values in a texture called Depth map. In the second pass we render the scene from the camera point of view and, by comparing the depth of the current fragment from light perspective with the depth saved in the Depth map, we can compute which fragments are lighten up and which one are in shadow. Having this information we can decide to make the fragment in shadow darker and this creates shadows in the scene. This technique allows us to have precise shadows but the quality of the result is influenced by the dimension of the Depth map. In out project we used a 2048x2048 depth map and we also applied a 3x3 Percentage Closer Filtering. This filter makes multiple shadow map comparison per pixel and averages them together to obtain smoother and better looking shadows.

3.5 Skybox and Environment Maps

A skybox or cubemap is a large cube centered in the origin of the world that encompasses the entire scene and contains six images of a surrounding environment, giving the player the illusion that the environment he's in is actually much larger than it actually is. To setup our cubemap we have to load six distinct images that will be attached to each face of the cube.

Figure 2: Cubemap textures

A cubemap used to texture a 3D cube can be sampled using the local positions of the cube as its texture coordinates. Being the cube centered on the origin (0,0,0) each of its position vectors is also a direction vector from the

origin. This direction vector is exactly what we need to get the corresponding texture value at that specific cube's position. For this reason we only need to supply position vectors and don't need texture coordinates.

```
interp_UVW = position;
vec4 pos = projectionMatrix * viewMatrix * modelMatrix * vec4(
    position, 1.0);
gl_Position = pos.xyww;
```

Listing 1: Vertex Shader

```
colorFrag = texture(tCube, interp_UVW);
```

Listing 2: Fragment Shader

In our project we set up three different cubemaps loading and linking the corresponding textures. When the player switches room or game mode, the program swaps the textures and a new environment is shown.

3.6 HUD

The HUD or *Heads-Up Display* is the method by which we can display information as a part the game user's interface. To develop HUD in our project we used the FreeType Library[2], a set of functions that is capable of produce high-quality output (glyph images) of most vector and bitmap fonts. Glyphs are a monochromatic images with transparency, antialiasing, and no drop shadow that uses a mask to define its shape. The setup of the FT Library required to load, through a standard font file, all characters into a structure converting them into glyphs. The rendering of the 2D text is then achieved by passing to the function *RenderText()* what we want to be shown on screen as string, including custom variables. The function scans the string and renders each character independently by sending to the GPU the corresponding glyph.

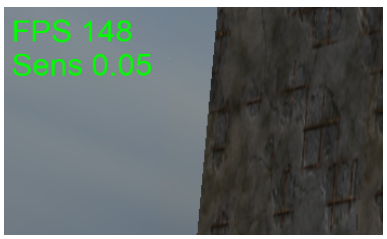

Figure 3: HUD Display

4 Second Room: Target grid

In the second room we kept the basic room structure (walls and buttons layout) but we used different textures, lights and lighting computation parameters. In this game mode are present 49 targets, placed in a 7x7 grid. All targets are rendered at the same time, but only one is considered active. The disabled ones are dark and does not emit light, while the active one is lighted up in red. We used metallic materials for the walls and the floor to emphasize the reflection

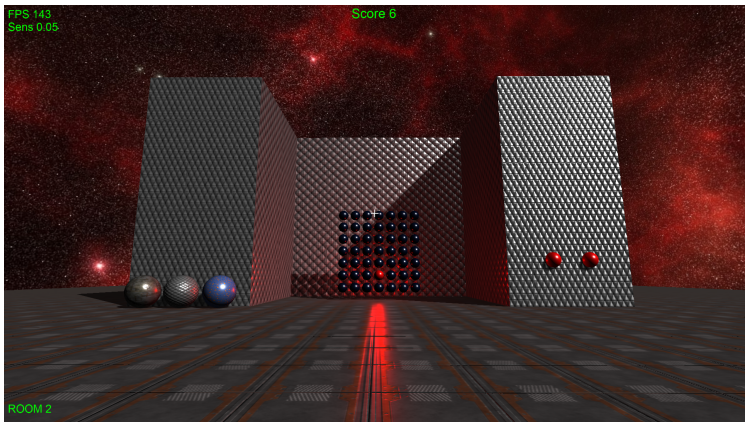

Figure 4: Second room

coming from the illuminated sphere.

```
1 if (active_targets[i-(walls_number+buttons_number)]){
2     ActivateTexture(5 + active_room * 6, 2.0f, textureLocation,
      nMapLocation, repeatLocation, proceduralLocation);
3
4     pointLightPosition = glm::vec3(body->getCenterOfMassPosition().
      getX(), body->getCenterOfMassPosition().getY(), body->
      getCenterOfMassPosition().getZ());
5 }
```

We use an array to keep track of which target is active. The sphere corresponding to the active index will be rendered using the active target texture (line 2) and the point light position will be the center of the sphere (line 3).

```
1 if (hit_sphere(center, radius, camera.Position)){
2     hit=true;
3     if(active_targets[i-(walls_number+buttons_number)]){
4         active_targets[i-(walls_number+buttons_number)] = false;
5         active = true;
6         score++;
7     }
8 }
```

```
1 if(active){
2     active_targets[(rand()%49)] = true;
3     active = false;
4 }
```

If the correct sphere is hit we set to false the corresponding index and a new random one will be activated.
We then update the spheres texture and the point light position in order to match the new active target.

4.1 Lighting computation: Point light

In the second room we added a point light in the position of the active target. Thus, we need to simulate reflection on the nearby metallic objects. The effect

is given updating the illumination parameters to $\alpha = 0.15$ and $R_f(0°) = 0.5$. In addition to the directional light, the final fragment color must take into account the contribution of all light sources.

```
1 vec3 finalColor = (1.0 - (shadow*0.7)) * (lambert + specular) *
      NdotL;
2 finalColor += ( GGX(surfaceColor, N) * 0.8 ) * pointLightColor;
3 colorFrag = vec4(finalColor, 1.0);
```

Listing 3: Fragment Shader

In the code above we add to the color computed considering only the directional light (line 1) the contribution of the point light (line 2). The point light contribution is computed using the GGX method seen in *subsection 3.1*. In the first and third room where the point light is not needed we turn it off setting it's color to (0,0,0).

5 Third Room: Bouncing targets

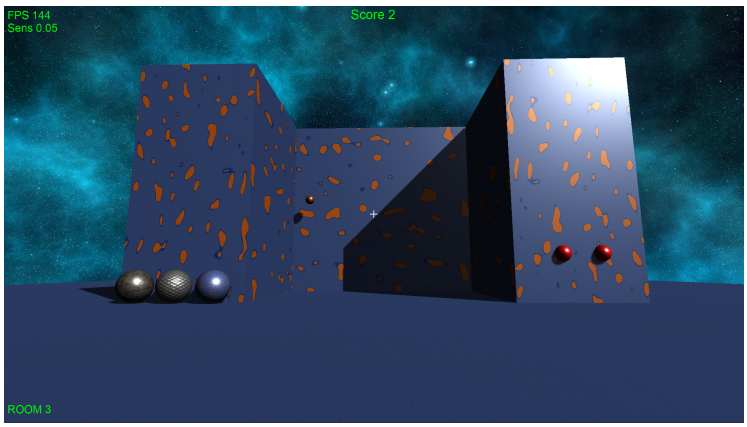

Figure 5: Third room

In the second room we kept the basic room structure (walls and buttons layout) but we used different textures, lights and lighting computation parameters. We used a rubber floor and a procedural texture for the walls. We also changed the lighting equation parameter to $\alpha = 0.2$ and $R_f(0°) = 0.9$. In this game mode there is only one target bouncing within the walls. Hitting the present one will make another one spawn in a random location. At the creation of the target a random impulse is applied to set it in motion and his movement is completely managed by the *physical simulation*.

```
1 if(hit){
2     target_pos = glm::vec3((num_side - (rand()%7)*0.5f)-5.5f, (
      num_side - (rand()%7)*0.5f)-2.5f, 0.7f);
3     reset_single_target(active_room,target_pos);
4
5     GLfloat shootInitialSpeed = 7.0f;
```

```
6       glm::vec3 shoot = glm::normalize(glm::vec3(randomNumber(-1.0f,
        1.0f), randomNumber(-1.0f, 1.0f), randomNumber(-1.0f, 1.0f)));
7       btVector3 impulse = btVector3(shoot.x,shoot.y,shoot.z) *
        shootInitialSpeed;
8       target->applyCentralImpulse(impulse);
9       hit=false;
10  }
```

In the code above we can see the sequence of actions executed when a new target is created. The position of the target is randomized in the same way as in the *first room* (line 2-3). The impulse generated as a normalized random vector with components between -1 and 1 to get different shooting directions (line 6). The impulse magnitude is decided by the *shootInitialSpeed* variable. Once the impulse is applied, the *physical simulation* handles the movement of the target.

5.1 Procedural Texture

A procedural texture is a texture created using a mathematical description rather than directly stored data. In our project we implemented a procedurally generated and animated texture to cover the wall of the third room customizing the code for a lava-lamp effect, created on the base of symplex noise[5]. The simplex noise is a noise function made by Ken Perlin. The symplex noise algorithm has lower computational cost with respect to the previous algorithms and avoids directional artifacts.

```
1   vec4 noise() {
2       vec2 u_resolution = vec2(0.2,0.2);
3       vec2 st = interp_UV*2/u_resolution.xy;
4       st.x *= u_resolution.x/u_resolution.y;
5       vec3 color = vec3(0.0);
6       vec2 pos = vec2(st*3.);
7       vec4 result;
8
9       float DF = 0.0;
10
11      float a = 0.0;
12      vec2 vel = vec2(timer*.1);
13      DF += snoise(pos+vel)*.25+.25;
14
15      a = snoise(pos*vec2(cos(timer*0.15),sin(timer*0.1))*0.1)
        *3.1415;
16      vel = vec2(cos(a),sin(a));
17      DF += snoise(pos+vel)*.25+.25;
18
19      color = vec3(smoothstep(.7,.75,fract(DF)))*vec3
        (256.0/256,110.0/256,15.0/256);
20      if (smoothstep(.7,.75,fract(DF))==1.0){
21          result = vec4(color,1.0);
22      }else{
23          result = vec4(1.0-color,1.0)*vec4
        (69.0/256,96.0/256,165.0/256,1.0);
24      }
25      return result;
26  }
```

The animation is created passing as input to the shader the execution time of the application using the uniform variable *timer*. This allowed us to update some parameters of the noise function at run-time generating the moving effect.

The original function produced a black and white pattern, so, as a further customization we also modified color output at line 20-24 to match the color pattern of the rest of the room.

6 Performance analysis

Performance were tested on 3 machines:

1. Intel Core i7-6700 CPU, Nvidia Geforce GTX1660 GPU and 16GB RAM
2. Intel Core i5-3570 CPU, Nvidia Geforce GTX1070 GPU and 8GB RAM
3. Intel Core i5-7300HQ CPU, Nvidia Geforce GTX1050 GPU and 8 GB RAM

For data acquisition we used *NVIDIA GeForce FrameView 1.2*, an application developed by *NVIDIA* for measuring frame rates, frame times, power, and performance-per-watt on a wide range of graphics cards. We run the program for 60 seconds (20 seconds for each room) and we got the following results:

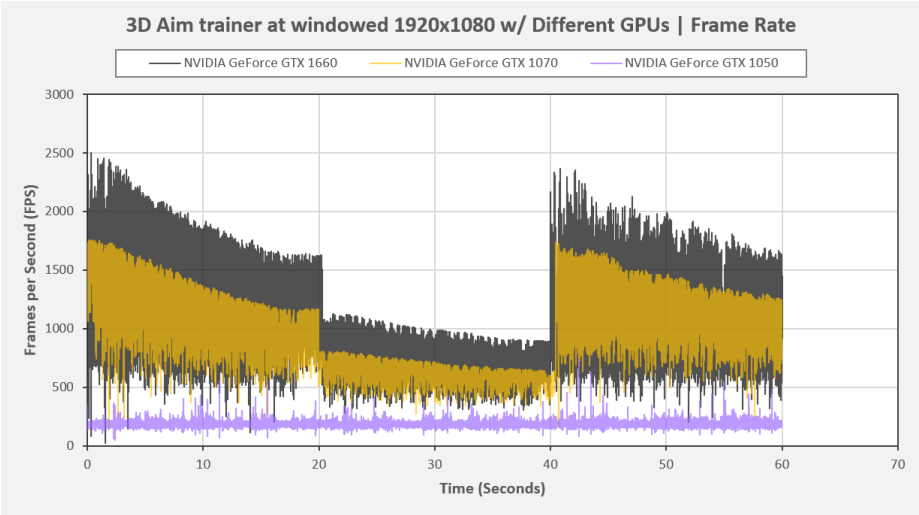

Figure 6: FPS comparison graph

FPS									
	Configuration 1			Configuration 2			Configuration 3		
Rooms	MIN	AVG	MAX	MIN	AVG	MAX	MIN	AVG	MAX
Room 1	25.8	1147.3	2457.0	189.9	957.6	1709.4	23.3	169.3	674.3
Room 2	211.9	711.9	1152.1	208.2	535.4	780.0	66.7	167.3	392.3
Room 3	27.6	1074.7	2409.6	144.7	1128.8	1675.0	70.1	192.7	600.2

Table 1: FPS comparison table

We can clearly see that the second room has a bigger impact on overall performances compared to the others, due to the higher number of models present in

the scene at the same time and a more complex light computation. We can also see that despite having similar GPUs in terms of power between first and second configuration we have a considerable fps drop, meaning that the CPU is a bottleneck. The third configuration presents very evident low performances due to lower level hardware. In this case we can notice a plain graph, meaning that the CPU is no longer a bottleneck, but the GPU is. We observe this through the fact that increasing the physical complexity of the scene in the second room does not result in a drop of fps.

7 Conclusions

Our goal was to create a real-time interactive application that could work as aim trainer. The final application results intuitive, fast and responsive as required by this kind of games. Collisions between bullet and target are precise and reactive and the overall performances are good even in the most populated scenes. We implemented three rooms, with different game modes and visual effects. Possible extensions of this project are the creation of other kind of game modes, with more complex models for the targets and more complex scenarios.

References

[1] *Bullet Library Documentation*. URL: `https://github.com/bulletphysics/bullet3/blob/master/docs/Bullet_User_Manual.pdf`.

[2] *FreeType Library Documentation*. URL: `https://www.freetype.org/`.

[3] *GLSL Documentation*. URL: `https://docs.gl/sl4/all`.

[4] *OpenGL*. URL: `https://www.learnopengl.com/`.

[5] *The Book of Shaders by Patricio Gonzalez Vivo & Jen Lowe*. URL: `https://thebookofshaders.com/11/?lan=it`.