**BEYSIM:**

**Real Time Graphics Programming Project Documentation**

By Filippo Giabelli (28582A)

**INDEX:**

0. MAIN IDEA

1. DESCRIPTION OF THE APPLICATION

1.1.First Sight Scene at Runtime

1.2. Interaction with User

1.2.1. Spinning Tops Interactions

1.2.2. Arena and Environment Interactions

2. IMPLEMENTATION

2.1. model.h & mesh.h

2.2. physics.h

2.3. particle\_system.h

2.4. Rendering Loop

2.5. io\_manager.h

2.6. Shaders

2.6.1. foreground\_shader

2.6.2. skybox\_shader

3. PERFORMANCE

**0. MAIN IDEA**

The main idea behind this project is to put in practice the concepts learnt during the Real Time Graphics Programming course, and some not covered by the lectures, by integrating them into a working real time application.

The application consists of a simulation of the launch of 2 spinning tops battling each other in a simple and small concave arena. This concept allows the project to involve not only rendering techniques, but basic physics simulation as well.

To demonstrate the knowledge of different rendering methods the spinning tops and the arena itself were provided of special graphical effects, such as sparks, flashing lights, etc.

**1. DESCRIPTION OF THE APPLICATION**

**1.1. First Sight Scene at Runtime**

The final application starts with an automatic launch of the two spinning tops, that will fall into the arena with an initial rotation. Firstly one top has a grey color, the other has a pinkn one, and the arena has a blue plain color and stands on a plane, to which a cosmic themed texture is applied. Everything is finally enclosed into a skybox, with a similar theme. The scene is illuminated by a far white light.

When the tops collide they produce sparks and a fast yellowish-white flashing light, until they have enough rotational speed.

**1.2. Interaction with User**

At this moment the application runs the physical simulation, and eventually ends when the tops stop. However if the user gives some specific inputs some more effects happen.

1.2.1. Spinning Tops Interactions

- Event: "1-key pressed": if the grey spinning top has at least a certain amount of rotational speed, it gets a little bit more, and it emits a green light, that turns off slowly within a few seconds, and some particles flowing in a vertical way for a little more brief period. If this event occurs it cannot be triggered again until a new launch (see "Enter-key pressed" event).

- Event: "2-key pressed": the same event as the one just described happens instead to the pink spinning top (the light emitted has now a pink color).

-Event: "Enter-key pressed": the spinning tops instantly get launched again, gaining their original angular velocity, but slightly translating by a random factor in order to have different interactions between each other. Now the two previous described events can be triggered once again.

1.2.2. Arena and Environment Interactions

-Event: "Spacebar-key pressed": the arena and/or the environment change, following this specific order:

-Now the arena reflects the skybox.

-Now the arena changes its color to brown, and the scene is filled with a moderately dense fog.

-Now the arena assumes a dark grey color, the main illuminating light is significantly softer and 4 changing color lights are turned on at the edges of the arena.

-The 4 lights are now turned off.

-The scene goes back to its original state (blue arena, and intense white environmental light)

If the events occurs again the scene changes again following the same order (from the top to the bottom).

**2. IMPLEMENTATION**

The main code starts setting up every library and every variable that will be useful in the later parts, and loading the models (see 2.1) and textures (via *texture\_loader.h*, capable of loading classic appearance textures as well as cubemaps).

The following subchapters will focus on describing the core algorythms, including the rendering loop, shaders, and the most relevant separate .h files.

**2.1. model.h & mesh.h**

These files manage the loading of models, creating geometry and index buffers, mesh by mesh for every model, and their rendering at runtime.

The model.h file implements the method *getBoundingBox()*, that return a 3D vector containing the dimensions of the tightest axis aligned bounding box of the model.

This method will be useful to resize the physical bodies for the simulation and to manage the collision detection.

**2.2. physics.h**

This module has the objective to manage the creation of a physics simulation and rigid bodies using the "Bullet" library.

Not only this module contains a function that returns a body that uses basic geometry shapes as meshes (*createRigidBody()*), but it also has one to create one from a mesh given as input (*createRigidBodyFromMesh()*).

**2.3. particle\_system.h**

This module manages all the particle systems. Its logic consists of a buffer of a predefined length, containing all the single particles, represented by a structure with all their attributes.

The method *OnUpdate()* is responsible of implementing the behaviour of every single particle through time, by updating its attributes depending on the time elapsed from frame to frame.

The method *OnRender()* instead renders every particle by creating a bidimensional square (that, being usually tiny, perfectly gives the impression of a particle, even in a 3D world).

**2.4. Rendering Loop**

In this while cycle, after the application stage, the core part of the logics is implemented.

First of all the physics simulation is started, then the static objects are drawn, to then end with the dynamics ones.

The two spinning tops are approximated as cones by the physics simulation system.

Furthermore, all the dynamic lights and particle system are updated.

To make the color of the "multiple lights" arena (if active) change in time, a sinusoidal function was used.

To generate sparks when the two tops hit each other, a worth to be explained algorythm was developed: first of all the two rays of the cones, the vector connecting the center of the cones, and the point being on the edge of one top being closest to the other top were all calculated. After that it's checked if the length of the vector connecting the two centers minus a tiny amount *epsilon* is less or equal than the sum of the two rays. If the test is passed, then the sparkles and flashing light are generated. The *epsilon* is to avoid calculation errors.

Finally the skybox is rendered, and the buffers are swapped.

**2.5. io\_manager.h**

This module is responsible to manage the behaviour of the application during the interaction with the user. This involves the events discussed in Chapter 1.2, as well as camera movement, not discussed there, but consisting of the classic "wasd" configuration, where "w" means "move to the position of the cursor" (elevation included).

**2.6 Shaders**

The current subchapter is a description of the shaders used in the application.

2.6.1. foreground\_shader

The vertex ang fragment shaders under this name are responsible of describing the logics behind the illumination of the majority of the objects in the scene.

To implement the variety of ways things appear a subroutine for the fragment shader was implemented. This set of functions tells the way to decide the final color of the current fragment. The most used is *GGX*, that uses in fact the GGX equationm applied to the arena (apart from when it reflects the skybox), the ground and the two tops.

When the arena needs to reflect the skybox, for its rendering the subroutine is set to *Reflection*, which computes the color of all of its fragments by casting a ray from the camera, computing the reflection ray, and use this last one to sample directly from the skybox, picking up the color it ends up on. The result color eventually adds to the GGX illumination model as the Lambert component, so that environment light is reflected as well (as the specular component).

To compute the sparks there was no need to take into account none of the lights, given their dimension and the fact that they are considered as many tiny sources of light. For this reason a *Particles* subroutine was implemented: it just returns a color passed as uniform (taken from the color attribute of the particle structure).

Finally, in the main function of this fragment shader, if a flag, passed as uniform, is set to true, the final color is computed in a slightly different way: the application is in the "fog" state, so, after computing the color in the regular way, it is modified to simulate fog in the current way:

*final\_color = d\*fog + (1-d)\*color*

where "fog" is the color of the fog and "color" is the color of the fragment without fog, and "d" is a number representing the distance of the fragment from the camera (accurately scaled and clipped to 1.0). Basically it's an interpolation between the two colors depending on the distance from the camera.

2.6.2. skybox\_shader

These vertex and fragment shaders were used to render the skybox. The logics is actually just applying to the final color of the fragment the color sampled from a *samplerCube* containing the properly loaded texture.

**3. PERFORMANCE**

The performance was tested on a AMD Ryzen 7 5700G processor, with a GeForce RTX 2060 GPU, and the maximum usage of CPU detected was 3.1% (extremely rare), with an average of around 0.8%-1.0%, while the FPS were constantly 60. The memory usage was around 100MB.

The application was tested on another, way elder, machine as well: an Asus laptop provided with Intel Core i7-8550U CPU (integrated GPU).

Here the application took roughly 1.7% of the CPU, and the FPS were floating around 25-30 (not bad considering the hardware).

The application in general isn't very hardware demanding, and the results confirmed this.

*(The "demo.mp4" stats window on the left refers to the global performances, taking into account background processes too)*