Magic Eight - User Doc

Phillip Stranger - 11807773
Dominik Völkel - 11811035

June 12, 2022

1 Setup & Execution

The game is compiled for Ubuntu (Linux).

1.1 Needed Libraries

To run the game, the libraries *allegro5* and *spdlog* need to be installed on the machine. As a prerequisite the community repository (universe) needs to be enabled on ubuntu. A tutorial can be found here: https://www.linuxshelltips.com/enable-universe-repository-ubuntu/

The following commands can be used to install the libraries *allegro5* and *spdlog*:

```
sudo apt update

#install allegro
sudo add-apt-repository ppa:allegro/5.2
sudo apt-get install liballegro*5.2 liballegro*5-dev

#install spdlog
sudo apt install libspdlog-dev
```

For more information on the libraries visit the links below:
allegro
allegro wiki
spdlog

1.2 Execution

To make execution easier we provide a shell script called *start.sh* situated in the root of the submission folder. This script links *./res* for the game (as this link is needed) and starts it afterwards. So to execute the game one has to start a terminal inside the root of the submission folder and call the shell script as such:

```
./start.sh
```

2 Game Manual

2.1 Controls

Left Mouse-Button	Fire White Ball
Right Mouse-Button	Spawn 4 Gravity Wells
ESC	Exit Game
P	Pause Game
R	Reset Game to initial state
S	toggle Settings

Table 1: Game Manuals

Table 1 gives an overview of the manuals of the game. The white ball is the only object which can be moved by the player. If the white ball is stagnant (i.e. not moving) it can be fired via clicking with the left mouse-button onto the table. The white ball then is shot into the direction of the mouse pointer and its velocity is determined by the distance between the white ball and the mouse pointer. The further away the click is from the white ball the faster the white ball is accelerated. As a visual aid we draw a red line away from the center of the white ball to the mouse pointer showcasing the direction and power of the shot.

By clicking the right mouse-button one can spawn 4 gravity wells, which act upon the billiard balls for 5 seconds generating chaos on the table. After 5 seconds they despawn again, as otherwise the game might be unplayable. Three out of the four gravity wells are spawned at a random position on the table and the last one is spawned at the position, where the right-click occurred.

2.2 Settings

We provide a wide range of adjustable settings, which can be adjusted in the settings windows, which is situated on the left side of the screen. This window can be resized by dragging the low right corner. Additionally this window can be toggled via pressing the key 'S'. Due to the bug, that the toggling of the settings window (pressing the 'S' key), as first input of the game, can somehow mess with the input manager, we opted for the small setting window, which can be resized by dragging the low right corner opposed to a full sized settings window at the start of the game. The adjustable settings are listed in table 2.

Things to keep in mind:
The FPS counter also affects the rendering of the GUI.
Lowering the PPS (physics-updates per second) too much will make the system very unstable and increase the likelihood of objects clipping into each other, as we calculate with delta time since last physics update. This means that at sub 30 PPS the system might crash as some objects are flung out of the play-space into *nan*.
Regarding the variable h for the Physically-based Particle Dynamics, as our physics manager uses delta time, h needs to be very small or else the calculation will have no impact.

Time Delta Multiplier (TDM)	Slider to adjust the speed of the physics simulation
Enable Debug View	Enable/Disable Debug View (Visualizations)
Enable Force-field	Enable/Disable Force-field Visualization
Table Friction	Enable/Disable Table friction
Enable Voronoi Noise-Overlay	Enable/Disable Noise-Overlay for Voronoi
Euler	Use Euler integration
Kutta	Use RK-4 integration
FPS	Slider to adjust the frames per second
PPS	Slider to adjust the physics-updates per second
h	Slider to adjust step size h
Ball Power (BP)	Ball Power multiplier
Num Voronoi Cells	Number of voronoi cells for voronoi diagram
Gravity Well Despawn Time	Time until spawned gravity wells are despawned

Table 2: Game Manuals

3 Gameplay

The game itself is a simple turn-based version of pool between two players with additional features, which make the game more difficult. The features are all correlated to the implemented animation techniques. Here we touch only briefly on the techniques, for more information refer to the *Techniques* section below 4.

First there is a big occluder object on the left lower side of the table making it way more difficult for balls to make their way to the left lower hole. This occluder object is made out of ice, so if it is hit by a ball it can crack (Voronoi Fracture).
Further we thought a normal white ball makes the game too easy, so we made it significantly harder to accurately pocket balls by surrounding the white ball by an ensemble of satellites (Hierarchical Transformation), which make it harder to play balls precisely, as they can interfere severely. Also with these rotating objects around the white ball shots need to be timed to make sure the satellites don't interfere with the shot one wants to make. As these satellites should only interfere with the balls, they only collide with balls (i.e. there is no collision between satellites and rectangles).
Lastly, by clicking the right mouse-button one can spawn 4 gravity wells, which exert a gravitational pull on the balls reshuffling the arrangement of the balls. As it would not be possible to play the white ball, if it would be affected by the gravitational pull, we exclude the white ball from the gravitational pull to make sure the player can still play the game, while the gravity wells are present.

The goal is to put all the balls into the holes and finish-up with the black ball. There are balls in two colors signaling to which player they belong. Yellow balls belong to Player 1 and pink balls belong to Player 2. In the top right corner we show the current ball count for each player and whose turn it is. If a Player pockets all balls (with the black ball being the last one) the player wins and it is displayed in the top right. The game can then be restarted via pressing the 'R' key or exited via pressing the 'ESC' key.

Bugs

Toggling the settings window (i.e. pressing the 'S' button) can mess with the input manager and thus it might not receive mouse-button-clicks. If it is not possible to shoot the white ball or spawn gravity wells, just toggle the settings window again or restart the game.

As stated above it is possible for balls to clip into objects, if the update rate of the physics manager is not fast enough to handle the collision before the ball is already inside the object. Additionally it is possible that this happens due to one of the satellites, traversing the white ball, pushing another ball into an object, when the ball has no way of escaping the pressure of the satellite ball. If a ball gets pushed out of the table (i.e. it escapes through the table border) the ball is handled as if it was pocketed in a hole.

4 Techniques

In the following we present the techniques we implemented, how we implemented them and how they are visualized/can be visualized.

4.1 Rigid-Body Dynamics

As pool depends on rigid body collisions *Rigid-Body Dynamics* was the first technique we implemented.

4.1.1 Implementation

We implemented the rigid body dynamics in the files *src/GameObjects/RigidBody.cpp, include/GameObjects/RigidBody.hpp*. There we handle the collision between rigid bodies. The bodies we handle are circles (balls) and rectangles (borders, occluders, etc.). Therefore we handle ball-ball collisions and ball-rectangle collisions. The check if a collision occurs is handled by the physics manager and if a collision occurs the RigidBody-class handles the collision. We handle the ball-ball collision via 2D elastic collision in an angle-free representation. The equations to calculate the new velocities are shown in Figure 1 and the information on how to handle the ball-ball collision were taken from [5, 4, 2]. Additionally we multiply the newly calculated velocities with a restitution factor taking into account the loss of power absorbed by the balls during the collision.
The collision between balls and rectangles is handled like a light reflection, as a ball bounces off a plane in the same manner as light, with incoming angle equaling outgoing angle. Therefore we use the equation for the reflection vector from the Phong reflection model to calculate the new direction of the velocity vector, which is as follows:

$$R = 2 * (L * N)N - L$$

, with R being the reflection vector, L being the incoming ray and N being the plane of the normal. We utilize this by calculating the normal of the plane with which the ball collides, this is our N. Further we use the velocity vector of the ball, which is our L. With this we can calculate the direction of the velocity vector after the collision and additionally we include the restitution factor of the collided rectangle to account for the loss of energy (loss of velocity).

$$\mathbf{v}_1' = \mathbf{v}_1 - \frac{2m_2}{m_1 + m_2} \frac{\langle \mathbf{v}_1 - \mathbf{v}_2, \mathbf{x}_1 - \mathbf{x}_2 \rangle}{\|\mathbf{x}_1 - \mathbf{x}_2\|^2} (\mathbf{x}_1 - \mathbf{x}_2),$$
$$\mathbf{v}_2' = \mathbf{v}_2 - \frac{2m_1}{m_1 + m_2} \frac{\langle \mathbf{v}_2 - \mathbf{v}_1, \mathbf{x}_2 - \mathbf{x}_1 \rangle}{\|\mathbf{x}_2 - \mathbf{x}_1\|^2} (\mathbf{x}_2 - \mathbf{x}_1)$$

Figure 1: Used equations for ball-ball elastic collision. Image taken from [2].

4.1.2 Visualization

All the required visualizations can be activated via activating the *Debug View* in the settings. The momentum vectors are visualized as black lines and the collision points are marked with red points.

4.2 Voronoi Fracture

We use an occluder object which is made out of ice and if a ball collides with it the ice breaks via a voronoi fracture.

4.2.1 Implementation

We implemented the *Voronoi Fracture* in the files *src/math/VoronoiFracture.cpp, include/math-/VoronoiFracture.hpp*. If a ball collides with a rectangle made out of ice it fractures. First we calculate the centers of the voronoi cells, which we calculate at random. After we have the cell centers we iterate over all pixels, which belong to the rectangle and assign them to the nearest cell. As a distance measure we use euclidean distance. Due to the fact that we support rotated rectangles we calculate the voronoi fracture on the unrotated rectangles (as with this it is easier to get the pixels of the rectangle) and rotate everything back afterwards. After calculating the voronoi cells and their pixels we need to visualize the cells. As the *allegro*-library, which we use for rendering, is a very simple and bare-bones library we need to visualize the cells by calculating the convex hull of each cell and drawing this convex hull. To calculate the convex hull we utilize the "gift-wrapping" algorithm (Jarvis march) [3]. The implementation of the convex hull algorithm is based on the implementation found at [1]. Further one can activate the noise-overlay for the Voronoi fracture by enabling the checkbox *Enable Voronoi Noise-Overlay* in the settings window. The noise function we use is based on the *perlinNoise*-function found in the lecture slides [6] and the *vecRandom*-function we use is based on the pseudo random *rand2D*-function found at [7].

4.2.2 Visualization

The voronoi cells are visible as soon as the ice occluder is fractured. To visualize the cell centers as well as the distance field one has to enable the *Debug View*.

Disclaimer:
When activating the noise-overlay it can happen that the convex hull algorithm fails for small cells, as they might loose too many pixels due to the noising, so that the calculation of the convex hull is not possible anymore. E.g. a cell with 4 pixels looses 2 pixels due to the noising - it is not

possible to calculate the convex hull of 2 points. If this happens just reset the game and re-fracture the ice block.

4.3 Hierarchical Transformations

We use hierarchical transformations to simulate a set of satellites traversing the white ball. More specifically we have a satellite which traverses the white ball. This satellite itself is traversed by a satellite, which again has its own satellite. So altogether we have 3 layers of satellites. The first layer rotates around the white ball. The second layer rotates around the first layer satellite and the third layer rotates around the second layer satellite. So we have a satellite rotating around another satellite, which also rotates around a satellite which rotates around the white ball.

4.3.1 Implementation

We implemented the *Hierarchical Transformations* in the files *src/GameObjects/SceneHierarchy.cpp, include/GameObjects/SceneHierarchy.hpp*. Here we build the *SceneGraph*, which consists of nodes. Each node has a pointer to the parent, a pointer to the game-object itself and a vector of pointers to its children. This structure allow us to quickly add nodes and to update the nodes accordingly. The main function in this class is the *updateVelocity*-function where we update the velocity of a node. As all satellites should move according to their parent their velocity is calculated as a sum of their own velocity and the velocity of their parent. With this we update all satellites iteratively and their velocities account for the movement of their parent object, leading to the satellites moving relative to their parents.

4.3.2 Visualization

The hierarchical transformations are visible at all times, as the satellites rotate around the white ball and are an fundamental feature of the game.

4.4 Physically-based Particle Dynamics

We use Physically-based Particle Dynamics to simulate 4 gravity wells, which act upon the billiard balls and introduce chaos on the pool table and rearrange the position of the balls on the table. As stated above, 3 out of the 4 gravity wells are spawned at random positions on the table and the 4th gravity well is spawned at the right-click-position.

4.4.1 Implementation

We implemented *Physically-based Particle Dynamics* in the files *src/GameObjects/ParticleDynamics.cpp, include/GameObjects/ParticleDynamics.hpp*. Here we implemented the functionality to calculate the velocity of an object to the gravity-well. This velocity to the well is calculated by either using Euler- or RK-4-integration. The call to apply the force of each gravity well on the game balls is given by the physics manager. As stated earlier the white ball is excluded from the gravitational pull of the wells. This means the gravitational pull only acts upon player balls and the black ball.
In the settings window one can change between the two integration techniques and also the variable h as well as the despawn time can be adjusted. As a standard, Euler integration is activated, but can be switched to Kutta (RK-4).

4.4.2 Visualization

When right-clicking onto the table 4 gravity wells are spawned, which are visualized as orange circles. Further the visualization of the force-field can be activated by checking the *Enable Forcefield* checkbox and the visualization of the object paths can be activated by checking the *Enable Object path* checkbox.

References

[1] Convex hull sample implementation. https://rosettacode.org/wiki/Convex_hull. Accessed: 2022-06-10.

[2] Elastic collision. https://en.wikipedia.org/wiki/Elastic_collision. Accessed: 2022-06-10.

[3] Gift wrapping algorithm. https://en.wikipedia.org/wiki/Gift_wrapping_algorithm. Accessed: 2022-06-10.

[4] Chad Berchek. 2-dimensional elastic collisions without trigonometry. https://www.vobarian.com/collisions/2dcollisions2.pdf. Accessed: 2022-06-10.

[5] Reinhold Preiner. Physically-based animation. https://tc.tugraz.at/main/pluginfile.php/748399/mod_resource/content/0/6.%20Physically%20Based%20Animation.pdf?forcedownload=1. Accessed: 2022-06-10.

[6] Reinhold Preiner. Voronoi fracture. https://tc.tugraz.at/main/pluginfile.php/753190/mod_resource/content/0/7.%20Voronoi%20Fracture.pdf?forcedownload=1. Accessed: 2022-06-10.

[7] Thorsten Renk. From random number to texture - glsl noise functions. http://www.science-and-fiction.org/rendering/noise.html. Accessed: 2022-06-10.