Documentation - Submission Final Game - Globby Warrior

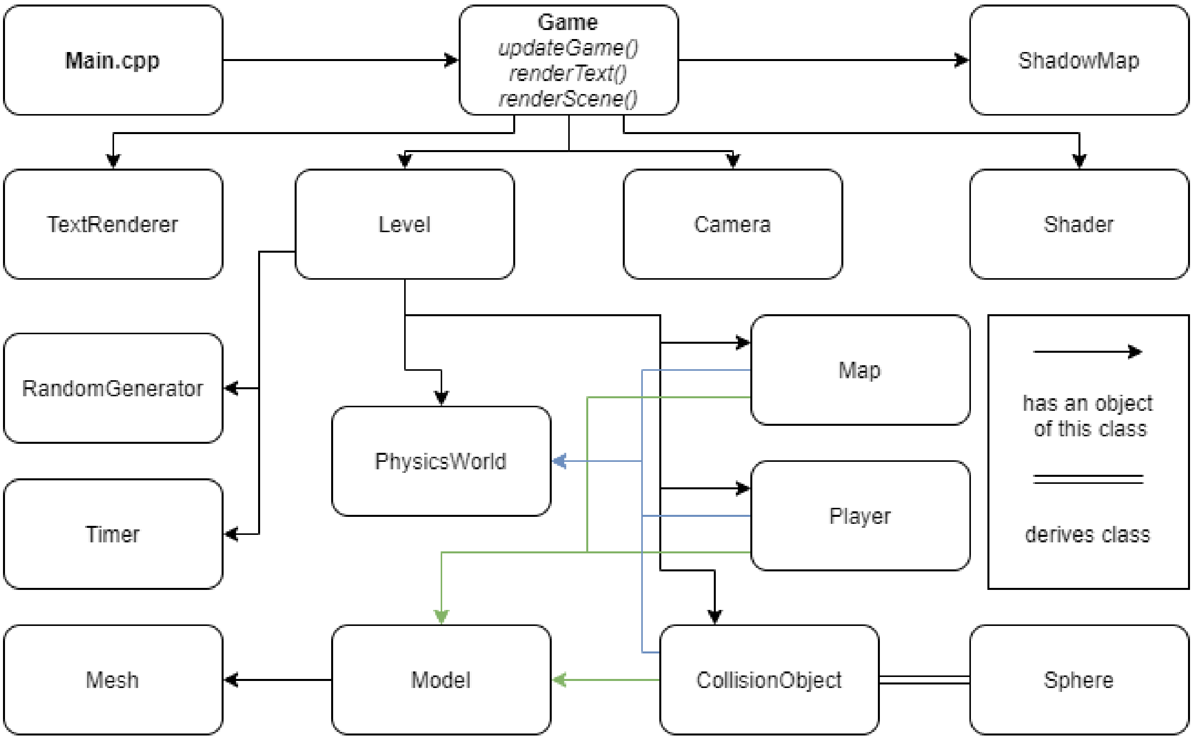
Benedikt Weber - 01627753
Maria Mußner - 51870828

Repository: https://github.com/mmaria495/cgue21-GlobbyWarrior.git

Project description

The idea of our game is that the treasure hunter Globby needs to survive a dangerous chamber without getting hit by bouncing spheres. To avoid getting hit, the player is able to move around and jump in the scene (W, A, S, D, SPACE). Additionally the camera view can be changed by clicking and holding the left mouse. To quit the game press ESC or close the window. Win and lose conditions are integrated as follows: to win the game the player has to avoid the spheres until the time has run out. The countdown can be found at the upper left corner of the window. If the player gets hit by a sphere, the game is lost.

Our scene models (*.obj files) are loaded via the object loader assimp, this includes the textures of the models.



Hierarchical structure of classes

The Game class handles all states of the Game including “menu, started, lost, won”. Depending on the current status the scene is rendered or not. Further, the text output of the window is generated (e.g. “You win!” or “59” for the timer - using the TextRenderer class). The Game object receives the models to be drawn from the Level object. In the Level class, the scene objects (collision objects, player and map) are set.

Additionally, we generate a PhysicsWorld object in the Level class, which is passed to every object of the scene in particular CollisionObjects, Player and Map. These scene objects generate rigidbodies for collision detection. The collision of Globby with one of the Spheres is checked in the PhysicsWorld class and passed to the

BulletObject struct of Level.cpp. Moreover, onUpdate() in Level.cpp manages all object actions (movement, spawning, drawing, movement…) of the scene. The actual rendering-call is implemented in the Mesh class.

For the physical properties of the game, the library Bullet was imported into the project. The different parameters and attributes (position, gravity, restitution, velocity,...) of the scene objects are stored in the according classes. To provide unpredicted conditions for our game, we are using a self implemented randomizer in the RandomGenerator class for spawning spheres.

For the scene lighting a directional light and a point light source are implemented. The lights are set in the Game class.

References and libraries:

Modelloader:

- assimp: https://www.assimp.org/
- assimp tutorial: https://learnopengl.com/Model-Loading/Assimp

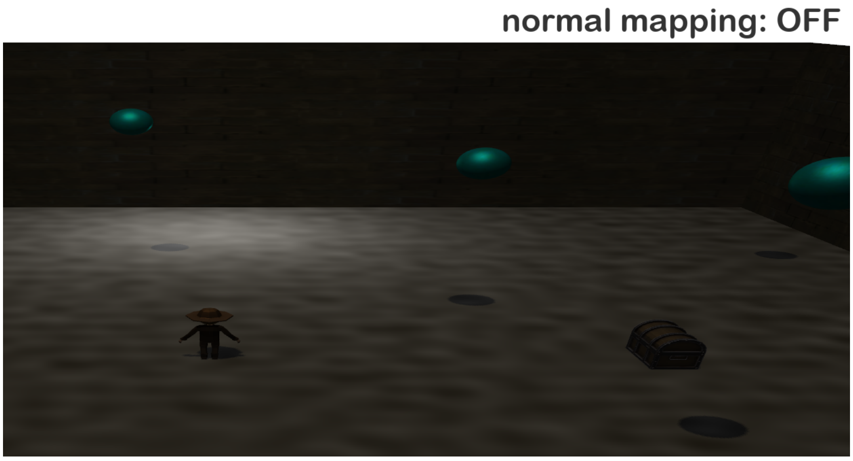
Text Rendering:

- freetype: https://sourceforge.net/projects/freetype/files/
- freetype tutorial: https://learnopengl.com/In-Practice/Text-Rendering

Bullet:

- imported as described in “Physx & Bullet Tutorials”
- bullet tutorials: https://tuwel.tuwien.ac.at/mod/book/view.php?id=1085446&chapterid=3850 several youtube videos
- collision detection: https://andysomogyi.github.io/mechanica/bullet.html

Effects:



Standard output (Procedural Texture: on, Shadow Map: active, Simple Normal Map: off)

- **Procedural Texture (8 Points)**
    - http://learnwebgl.brown37.net/10_surface_properties/texture_mapping_procedural.html

  In the above tutorial procedural textures for WebGL are described. We modified these for our shaders in OpenGL. Additionally we adjusted the colors and the noise, so that the texture fits our scene. We use a procedural texture for our ground plane (see screenshot above). The implementation can be found in assets/shader/ground.vert and assets/shader/ground.frag.

- **Shadow Map with PCF (16 Points)**
  - https://www.youtube.com/watch?v=9g-4aJhCnyY
  - https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping
  - https://github.com/VictorGordan/opengl-tutorials/tree/main/YoutubeOpenGL%2025%20-%20Shadow%20Maps%20(Directional%20Lights)

To add shadows to our scene we first generate the shadow map from the directional light perspective. Therefore we implemented a ShadowMap (ShadowMap.h and ShadowMap.cpp) class which is responsible for the respective render calls as well as for setting up the texture properties. To generate a shadow map we also had to create 2 additional shaders called assets/shader/shadow.frag and assets/shader/shadow.vert.

Secondly we can render the scene and add the computed shadow texture to the shadow calculations of the other shaders: texture.frag, texture.vert, ground.frag and ground.vert. There the shadows are added to the according fragment position. To avoid shadow acne a pcf function was applied.

- **Blobby Object using Marching Cubes (16 Points)**

The idea was to generate a voxel grid describing the whole scene and calculating the density values of Meta Balls based on this field functions:

$$F(x) = \sum_{1}^{n} f_i(||x - c_i||) \qquad f_i(r_i) = a(1 - \frac{4r^6}{9b^6} + \frac{17r^4}{9b^4} - \frac{22r^2}{9b^2})$$

(x = grid point, c = centroid/center, r = distance, a = amplitude/max influence, b = blobbiness/range of influence)

References of the functions can be found here:

https://people.cs.clemson.edu/~dhouse/courses/881/notes/metaballs/index.html

http://paulbourke.net/geometry/implicitsurf/

Instead of the Meta Ball calculation of Paul Bourke's article, we are using the Soft Object function to get a slight advantage considering computational costs. In the following, we are using Metaball as a more generic term for Blobby/Soft Objects.

Implementation description (unfortunately we were not able to finish our Marching Cube implementation - further explanation in last paragraph):

With regard to our map size the grid boundaries are set to min(-20,0,20) - max(20,20,20). We also set a granularity variable to 8 which causes step size of 0.125 and gives us an overall cube number of $(40 * 8 - 1)^2 * (20 * 8 - 1) \sim 16\,180\,000$.

To decrease the amount of cubes we have to march through per render call we integrated bounding boxes. We set a maxRadius to 2 (Metaball radius was meant to be 0.5 resulting in approximately 512 cubes per blobby object) which ends up in max 512*4 cubes per object. This maxRadius was set static just for testing computational cost. The actual bounding boxes would have been dynamic: 1) calculate the distance of 2 Metaballs where merging starts to take place (since all Metaballs have the same size this could be done in the constructor of the grid) 2) recursively calculate the distance of the centroids of the Metaballs and 3) for those with distance < mergeDistance add mergeDistance/2 to maxRadius and ignore the second Metaball during render call 4) else set maxRadius to mergeDistance/2…

In combination with some additional efficiency improvements (OpenMP calls, memory management before calling render loop, only marching through neighbors of intersected cubes,...) we were hoping for an efficient (60fps) implementation on the CPU. Current tests of the density calculation of the scalar field within the generated bounding boxes of 8 Metaballs prove us wrong resulting in 20fps (scalar field calculation only → without rendering of meshes).

Further readings regarding Marching Cubes algorithm:

http://paulbourke.net/geometry/polygonise/

http://users.polytech.unice.fr/~lingrand/MarchingCubes/accueil.html

http://docplayer.org/49306118-Marching-cubes-erstellung-von-polygonmodellen-aus-voxelgittern.html

https://mshgrid.com/2020/02/03/implementing-your-own-metaballs-and-meta-objects/

Code examples:

https://github.com/dgr582/article-marching-cubes

http://www.paulsprojects.net/metaballs2/metaballs2.html

(The current implementation can be found on the MarchingCubes branch. Note that we are currently also facing some issues initializing the grid. The grid might even be too big for the heap we set up - or we did something wrong declaring the heap memory.)

So, we started investigating GPU implementations, but due to time limitations and also a lack of code examples had to pick Simple Normal Mapping instead. Readings for GPU implementation:

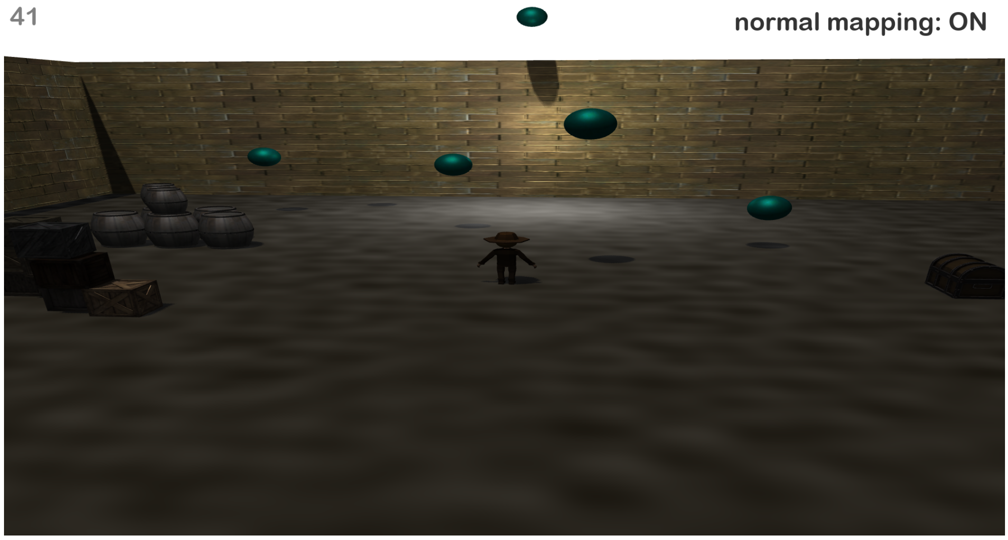
https://journal-bcs.springeropen.com/articles/10.1007/s13173-012-0097-z#:~:text=Marching%20cubes%20is%20one%20of,of%20auxiliary%20spatial%20data%20structures

http://s3.amazonaws.com/arena-attachments/2340738/f399ec0b3980a790f3752a8897dba9a8.pdf?1529596185

https://core.ac.uk/download/pdf/48548176.pdf

https://learnopengl.com/Advanced-OpenGL/Geometry-Shader

- **Simple Normal Mapping (4 Points)**



Scene with activated normal mapping (activated via right mouse)

  - https://learnopengl.com/Advanced-Lighting/Normal-Mapping
  - https://learnopengl.com/Lighting/Multiple-lights

For simple normal mapping we integrated one additional point light to the scene. This point light is placed to the opposite wall of Globby. The x coordinates of the character and the point light are matching, so the light moves according to Globby. Simple normal mapping can be turned on/off with a right mouse click. Per default it is turned off but you can also check the current state on the upper right corner. To add simple normal mapping to our game we had to adjust our assimp Model loader and duplicated texture.vert/frag. In the new shaders normal.vert/frage we added a sampler for the normal map and ignored all per mesh normals previously loaded with assimp.