Marbulous

Gameplay

The gameplay of Marbulous is mostly finished. Of course there is always room for improvements but the core gameplay is pretty solid. The player can move the marble in all directions and jump. The goal of the game is to finish a level as fast as possible. The timer starts as soon as the player moves and finishes when the player touches the white-blueish crystal in a level. There are also collectible jewels which reduce the timer by 1 second. When the player finishes a level his time is printed in the console and the records file /res/levels/records.json is looked up. If the player has set a new record the new time is stored and a message is printed to the console.

Vertical walls can be “climbed” by pressing the marble to the wall and jumping rhythmically (necessary to complete level 4). The trick here is not to move down. As soon as the marble moves down climbing does not work anymore.

Effects

A very helpful tutorial website for all effects was http://learnopengl.com/.

Shadow Mapping with PCF

Currently Shadow mapping with PCF is implemented. The original plan was to implement variance shadow mapping. While implementing we ran into some issues which we could not resolve. We will try to fix these until the Game Event.

Normal Mapping

Normal mapping was implemented. Every model in the game uses a diffuse, specular and normal map, mostly generated with CrazyBump. For normal mapping the tangent and normal vectors provided by Assimp were used. The bitangents were calculated out of these two vector because the bitangent vectors provided by Assimp are sometimes smoothed/interpolated which causes them to be not exactly normal to each other.

Bloom

Bloom was implemented with a FBO and two color attachments. The first stores the usual picture, the second only values above a threshold. Afterwards the second texture gets blurred with the help of ping pong buffers and a Gauss kernel divided into two shaders (horizontal and vertical). To improve the blur the ping pong buffers have a lower resolution than the original texture. This improves the blur one the one hand because of linear interpolation of the graphics card and also saves a lot of processing power.

1325696 Esberger, 1327434 Fast

Adaptive HDR

The textures attached to the FBOs use the GL_RGB16F format for increased accuracy and to prevent value clamping. For the initial color texture (before post processing, bloom) mip maps are generated. After that a quad will be rendered to a 1x1 pixel texture which gives the average color of the scene. This value is then sampled in the in the bloom/HDR shader which combines the three textures (usual scene, blur, luminance texture). In the shader the luminance is calculated and the value is then used for exposure control. After that tone mapping and gamma correction is applied to give the final image which will be rendered to the default FBO via a quad.

The adaptive HDR effect can be witnessed by turning around and looking to the “sun” (skybox).

Complex Objects

There are some complex objects in the game for example a statue in level 2 and 4 or the crystal at the end of each level. All models in the game are loaded via Assimp and the .obj file format.

Animated Objects

There are several animated objects in the game. In level 3 there is a platform which is the parent object of a spinner object. Therefore this is a hierarchical animation. The same concept is applied in level two but there the child object is only following the parent platform without its own animation.

View-Frustum-Culling

View frustum culling is implemented with simple spheres around objects which will be calculated out of the mesh data. The tutorial at Lighthous3d.com was used as a basis for the implementation. (http://www.lighthouse3d.com/tutorials/view-frustum-culling/)

Transparency

Transparency was used at some places in the game. Crystals and jewels are semitransparent. There is also an interactive transparency which will be activated for objects which are in the line of sight between the camera and the player. The objects in the line of sight are identified via ray casting with the Bullet library. To reduce artifacts all transparent objects are sorted by the distance from the camera. For the objects in the line of sight the intersection points of Bullet are used to increase the accuracy of the depth calculation. After the semitransparent objects are sorted every one of these objects will be drawn twice. First with front face culling and afterwards with back face culling. Overall this strategy works pretty well and is also pretty fast.

Experimenting with OpenGL

VBOs and VAOs are used to store and bind meshes for drawing. FBOs are used for post processing (Blur, Bloom, HDR), shadow mapping and in general to have better control over the render pipeline. Mip maps are used for texturing as well as the luminance estimation for HDR. The texture sampling quality can also be changed.

The controls are similar to the wiki:
F1 - Help (prints these controls)
F2 - Frame Time on/off
F3 - Wire Frame on/off
F4 - Textur-Sampling-Quality: Nearest Neighbor/Bilinear
F5 - Mip Mapping-Quality: Off/Nearest Neighbor/Linear
F6 – Anisotropic filtering qualities
F7 – Postprocessing (Bloom, HDR, Tone mapping, Gamma correction) on/off
F8 - Viewfrustum-Culling on/off
F9 - Transparency on/off

Controls

Controls for keyboard and mouse as well as joypad (PS4) were implemented:

- Player
  - WASD/Arrow keys: move in the given direction
  - Space/Right Ctrl: jump
  - Mouse: change orientation
  - Mouse wheel/Page up/down: change distance from player
- Ctrl: unlock the mouse
- F: first person camera
  - WASD: move around
  - Mouse: look around
  - Space/Right Ctrl: Move up
  - X/Right Shift: Move down
  - Left Shift: Move faster
- System
  - Escape: Close the game
  - Num1 – Num0: load level 1 to 10
  - R: reload shaders
  - C: clear models (reloaded at next level change)
  - Numpad0: show Bullet debug drawer
- Audio
  - Return: next song
  - B: volume +10
  - N: volume -10
  - M: mute audio

The joystick has the same actions on multiple buttons to counteract different controller layouts:

- Third person:
  - Left stick: Move player
  - Right stick: move camera
  - Button 0, 1, 4, 5: jump
  - Shoulder triggers: change distance to player
- Button 8: switch to first person camera
  - Left stick: move camera
  - Right stick: look around
  - Shoulder triggers: move up/down
- Button 9: reload level

Other stuff

Libraries

The following libraries were used

- SFML (Window context, Sound): http://www.sfml-dev.org/
- Assimp (Model loader: http://www.sfml-dev.org/
- Bullet (Physics engine): http://bulletphysics.org/
- jsoncpp (Level format): https://github.com/open-source-parsers/jsoncpp
- Blender (Modeling): https://www.blender.org/
- FreeImage (Texture loading): http://freeimage.sourceforge.net/

Level format

A level format was developed to make level creation easier. The main focus here was that it is human readable, easy to understand, flexible and extendible. For this purpose JSON was used. The levels can be edited while the game is running. Simply reloading a level (number keys) loads the new content. Also the models can be changed while the game is running via Blender. For this pressing the C key and reloading the level is sufficient. The level format enables content creation without any source code changes. Collision shapes are generated from the mesh data. Objects can easily be animated. Animations can have names assigned for later reuse with other objects.

Small things

A debug drawer for Bullet was implementet. The various modes can be accessed by repeatedly tapping the Numpad0 key. The window can be resized. The FBOs will be resized accordingly while running. Dynamic objects can be pushed of the platforms (pillars, statue). A skybox was used. The Game uses one global directional light source. Meshes and textures are instantiated once and used for all models. Smart pointers are used for memory management.