

Tower Jump

Background story

You have been captured by the Big Bad Boss and thrown into prison. You have found out that you can escape by making a series of difficult jumps. Can you make it out of the prison, or will you be stuck in here forever?

The main gameplay element is jumping. The character needs to jump onto several platforms to the top of the tower, where he can escape.

Current Gameplay / Controls

The character can be moved with WASD + SPACE. Note that while holding down SPACE the character “charges” a jump and will jump as soon as SPACE is released. The longer space is held, the higher the jump will be, holding down other movement buttons will also incorporate this movement into the jump.

Additional Controls (also listed on screen):

- F1: Toggle Wireframe mode
- F2: Toggle Backface Culling
- F3: Toggle Physics Debugging (Wireframe + AABBs). Note that if Wireframe and Physics debugging are combined, only the physics output is displayed
- F4: Toggle Free Camera
- F5: Toggle Info Text about Hotkeys
- F6: Toggle complete HUD
- F9: Reset game
- F11: Toggle Fullscreen

Description of the implementation

The main function in Main.cpp is rather straight forward. It loads all shaders, initializes the physics engine and the HUD and then loads the whole scene including objects, materials, textures, lights, the skybox, etc. using the AssimpLoader class. The render loop then polls the continuous input, simulates physics, handles basic gameplay, updates the so-called “peashooters” (the cannons shooting green goo) and then renders the game using a multi pass rendering pipeline (shadow calculation for all directional lights (just one present in this scene), a main render pass as well as post processing).

There is at least one class for every main feature in the game: shader loading is handled in the Shader class, material properties are applied by the Material class, there are multiple subclasses that utilize different textures. Textures themselves are loaded using the Texture class (and their subclasses for different loaders). All light types have

representations as objects of the Light class (and subclasses) and can be dynamically drawn by the game (only a maximum light amount has to be specified in the shader). This allows free placement of the lights in the blender scene since lights are also imported. The player is represented by the CameraPlayer class and includes the player physics as well as free moving camera, the HUD is also in a separate class. All physics features (game world, individual physics objects as well as debug drawing) is implemented in the Physics/ subfolder. For each Geometry (generated as well as loaded geometry, although we do not use all the generated geometry) there is a subclass of the Geometry class which handles the draw calls.

The whole scene, all lights, the geometries used in the scene as well as textures and materials are loaded from an FBX file (note that normal maps are loaded as emissive textures since blender refuses to export a reference to the normal map in the material, similarly PBR values are loaded as specular texture). After loading all objects and their textures, the assimp loader generates a list of objects to add to the scene. Such an object (Object class) is simply a container combining a Geometry object with a material and a physics object and giving it a name.

Notably we also implemented a post-processing pipeline in the PostProcessing class, its usage is demonstrated by the bloom implementation: a Pipeline houses two frame buffers plus respective textures and draws each texture to the other frame buffer in a flip-flop fashion for each shader present in the pipeline. The first execution is done using the input texture, the last texture that was filled is returned to the caller. This allows for an easily pluggable post-processing pipeline.

Features of the game

Compulsory Gameplay:

All features implemented

- 3D Geometry: Most of the scene is loaded from a .fbx file generated in Blender. For each point light, there is a glowing sphere added dynamically on import, Goo is also generated procedurally.
- Playable: the character can be moved with WASD+SPACE
- Advanced Gameplay: Sophisticated Jumping mechanism as well as moving obstacles (Goo)
- Min 60 FPS & Framerate Independence: the game is currently running below 16ms (above 60FPS) and framerate independent.
- Win/Lose Condition: The player wins by reaching the top level of the tower and loses if the timer runs out.
- Intuitive Controls: the movement of the player is like standard first-person games.
- Intuitive Camera: Present
- Illumination Model: Multiple light sources are loaded within the .fbx file and the objects have Materials (also imported).
- Textures: most objects have textures, only the glowing parts in the scene do not (point light balls + Goo)
- Moving objects: There are several cannons shooting balls of green goo across the tower. These vanish on impact but can kick the player away.

- Documentation: Present
- Adjustable parameters: The config file is present (settings.ini). Note that the brightness parameter is the HDR tone mapping exposure parameter, so it scales logarithmically.

Optional Gameplay:

- Collision Detection: Covered by the physics engine Bullet.
- Advanced Physics: There are several cannons shooting balls of green goo across the tower. These vanish on impact but can kick the player away. The vanishing as well as the win condition are implemented using collision callbacks.
- Heads-Up Display: Simple display of text with freetype, and a charging bar when jumping.

Effects:

- Lighting: Shadow Map with PCF: The main directional light (sunlight) projects shadows onto the scene. Shadow ACNE is fixed by applying a shadow bias, peter panning is dealt with by not using thin geometry, as advised in the revision course 2019 slides. PCF is performed as necessary.
- Advanced Modelling: CPU Particle System: The Goo is rendered using instancing. All Goo instances are collected in the PeaContainer, which also handles the lifetime of each Goo and drawing. Note that each Goo is also updated by the physics engine for the advanced modelling task.
- Texturing: Video Texture: The TVs at the top of the tower run a 4-second 30fps video. The frames are all loaded into memory at the start (see VideoTexture) and streamed to the GPU in real time. You can use the free camera to check out this feature without needing to complete the game.
- Shading: Simple Normal Mapping: We implemented this since it makes the whole scene look a lot better. We did not implement the on/off key, as we also implemented Physically Based Shading and are aware that this point does not contribute to our grade.
- Shading: Physically Based Shading: All objects except the glowing parts (light bulbs + goo) and the skybox are rendered using the Cook-Torrance model.
- Post Processing: Bloom/Glow: The Point lights (torches) as well as the Goo will glow. This is best observed in the wireframe view since we do not turn it off there and most of the other objects will not be shown.

The implementation for these tasks/effects follows the links referenced in the TUWEL course (<https://tuwel.tuwien.ac.at/mod/page/view.php?id=1085431>)

Additional Libraries

Bullet <https://github.com/bulletphysics/bullet3/releases>

Assimp <https://www.assimp.org/>

Freetype <https://www.freetype.org/>