Pac3D Documentation

Leo Preissegger (12122526), Enisa Azizovic (01228809)

13. Juni 2023

1 Feature Description

1.1 Collision Detection and Physics

Collision Detection is used to prevent the player from going throw walls and to detect when the player hits ghosts or coins. Collisions are reported to the code using `onTriggerEnter, onTriggerExit, onCollisionEnter, onCollisionExit` callbacks. In the game it is possible to shot projectiles at ghosts. To do so: new entities are created and a force is added to them. Another feature of the physics implementation is the use of raycasts. It is used to prevent the camera from going into walls and to check if ghosts can see the player, so they can chase him.

Following Colliders are supported: Box Collider, Sphere Collider, Capsule Collider, Trinagle Collider and Convex Collider.

Triangle Colliders and Convex Colliders must be cooked using the PhysX Cooking library. The meshes used to do so are loaded from AssetFiles, the same way how it's done for rendering.

Another feature of the engine that the game uses is a CharacterController. It can be used via the CharacterControllerComponent. A character controller is basically a kinematic rigid body with a few extra things. It cannot go throw walls and has a feature to automatically climb stairs, for example.

All of this is implemented using the PhysX library.

1.2 Heads-Up Display

The heads up display is implemented with a batch-renderer, that renders quads after the other parts of the scene have been rendered. For every UI primitive another shader is used. This shaders support outlining of circles and rectangles. Text rendering is done by using a font atlas that is generated at the beginning, with characters loaded from FreeType.

The batching is done by combining all elements of the same primitive type and the same z-index into a single Vertex Buffer and rendering them together. The distinction between z-indices is needed to archive proper transparency sorting.

Resources used to implement batch rendering: TheCherno - Batch Rendering (YouTube)

The HUD shows the amount of lives the player has left and how many projectiles he can shoot at the moment. Additonally there is a start screen, lose screen and win screen.

To configure The HUD the following component is used: UICanvasComponent

1.3 Shadow Map with PCF

The game supports shadows for objects in the world. This is implemented using Cascading Shadow Maps and PCF. For every frame 4 shadow maps are rendered (4 cascades) from the light point of view. Every cascade handles different parts of the camera frustum, to achieve better shadow quality and move the shadow maps with the camera movement. It is also possible to decide if shadows should be casted from any given object.

For now the game supports shadows for only one directional light.

1.4 Bloom/Glow

The Bloom implementation uses a technique known as Physically based Bloom. It works by down sampling the final image a few times. After that the smallest image is upsampled again and combined with the next larger image. This is done until we are back at the start resolution. Finally the bloom image is blended together with the scene image.

Articles describing the technique: Next Generation Post Processing in CALL OF DUTY: ADVANCED WARFARE, LearnOpenGL

1.5 Physically Based Shading

The renderer is using PPR for all geometry. It supports: albedo, roughness, metalness, emission and and a normal map. The before mentioned values can also be used from texture maps.

For lighting the following light sources are supported: Directional Light, Point Light, Spot Light and Image Based Lighting, more on that in 1.7.

Following resources were used to implement it: LearnOpenGL, Filament PBR

1.6 Normal Mapping

Normal mapping works by using data from a normal map instead of vertex normals. As the normal map is in tangent space, every vertex needs tangent and bitangent vectors. Using the tangent, bitangent and normal a TBN matrix is constructed, that is used to transform normals from tangent to world space. Normal mapping only works for imported assets, not for built in meshes, because they have no tangents and bitangents defined.

1.7 Environment Map

Environment mapping in the game is implemented by using image based lightning. To do this a HDRi texture is loaded and preprocessed. While preprocessing, two CubeMaps are calculated via Compute-Shaders for later use: Irradiance Map (Diffuse light) and Radiance/Prefilter Map (Specular light).

For the radiance map, different roughness levels are stored in different mip-Levels. The irradiance map is computed by convoluting the HDRi, because every pixel on the hemisphere has influence on the light at the fragment.

Following resources were used to implement it: LearnOpenGL

1.8 Animation

Animation can be done easily by updating the Transform Component TransformComponent every frame in the `onUpdate` callback.

It is also possible to load skeletal animation from Asset files. This can be done via the AnimatedMeshRendererComponent. The implementation works by loading all the Bones, their vertex weights and the animation data. Every frame the animation data is interpolated to compute the current state. The actual skinning is done on the GPU using a compute shader. We are using a compute shader and not the vertex shader, because we need the skinned data more than once per frame. In the compute shader a new Vertex Buffer is written, that is used in all subsequent operations.

Following resources were used to implement it: LearnOpenGL, OGDev

1.9 Adjustable Parameters

Application wide parameters can be set in `settings.ini`. Examples for parameters there are: resolution, fullscreen, refresh-rate, v-sync, anisotropic-filtering, shadow-map-resolution, startScene.

Scenes are created and loaded via XML Files. In this file the complete scene hierarchy can be parameterized. Some other things like Materials and PhysicsMaterials are also defined via XML files.

2 List of Implemented Gameplay and Effects

- Gameplay
  - Mandatory
    - 3D Geometry
    - Playable
    - Advanced Gameplay
    - Min. 60 FPS and Framerate Independence
    - Win/Lose Condition
    - Intuitive Controls
    - Intuitive Camera
    - Illumination Model
    - Textures

        * Moving Objects
        * Documentation
        * Adjustable Parameters
    - Optional
        * Collision Detection (Basic Physics)
        * Advanced Physic
        * Heads-Up Display
- Effects
    - Shadow Map with PCF
    - GPU Vertex Skinning
    - Environment Map
    - Simple Normal Mapping
    - Physically Based Shading
    - Bloom/Glow

All listed tasked were implemented in the game. Detailed descriptions for some of the tasks can be found in 1.

3 Implementation

Some particularly interesting aspects of the code are described here.

3.1 Entity Component System

Every object in the game is represented as an entity. Every entity has components attached to it, to store data about it. The structure of the scene can be loaded from a scene file on startup or when switching scenes. Entities and components can be dynamically added or destroyed.

This system is implemented without the use of any libraries.

All components available are described in section: 5.

Resources used: A SIMPLE ENTITY COMPONENT SYSTEM (ECS) [C++]

3.2 Scripting

To every entity a script can be attached via the ScriptComponent. These scripts are updated every frame via `onUpdate` and `lateUpdate`. There are also callbacks from the Physics Engine as mentioned in 1.1.

The Scripting system is heavily inspired by the Unity Engine.

3.3 Renderer

The renderer in the game has a full HDR Rendering pipeline. That uses a few render passes: ShadowMapPass, PreDepthPass, GeometryPass, SkyboxPass, PhysicsColliderPass, NormalsDebugPass, DebugLinesPass, BloomPass, ScreenPass, UiPass.

As one can guess many of the passes are just for debugging, but for that purpose extremely useful.

The PreDepthPass is used to generate a Depth Buffer without using a Fragment shader. This is done to prevent necessary overdraw in subsequent passes, to improve performance.

Every frame all objects that should be rendered are submitted to the Renderer to be processed later. This is useful, because with the information of all objects, the renderer can instance objects together.

4 Libraries

Following libraries were used to implement the above described features:

Name	Link	Usage
spdlog	https://github.com/gabime/spdlog	Logging in the Game (useful for debugging)
inih	https://github.com/benhoyt/inih	Reading of .ini files
glfw	https://github.com/glfw/glfw	Window management abstraction
glm	https://github.com/g-truc/glm	OpenGL Mathematics (GLM)
glad	https://glad.dav1d.de/	OpenGL extension loading
pugixml	https://pugixml.org/	Loading and parsing of XML files
stb_image	https://github.com/nothings/stb	Loading textures from files
assimp	https://github.com/assimp/assimp	3D Model loading
PhysX	https://github.com/NVIDIA-Omniverse/PhysX	3D Physics
FreeType	https://freetype.org/index.html	Font loading

5 Component Docs

Following Components can be used in the engine:

- TransformComponent
- MeshRendererComponent
- AnimatedMeshRendererComponent
- CameraComponent
- ScriptComponent
- DirectionalLightComponent
- PointLightComponent
- SpotLightComponent
- SkyboxComponent
- BoxColliderComponent
- RigidBodyComponent
- SphereColliderComponent
- CapsuleColliderComponent
- TriangleColliderComponent
- ConvexColliderComponent
- CharacterControllerComponent
- UiCanvasComponent

Component docs can be found here: GitHub