

Anemoi - Implementation documentation

Implemented feature	Short description, additional details, programming tricks, special features, additional libraries
Intuitive Camera	DebugCamera and 3rdPersonCamera both inherit from Camera. The original Camera class was not used, a new custom Camera was created. Both cameras are switchable by pressing '1', DebugCamera can freely fly and move. 3rdP-Cam is fixed and has lookat set on Player, but up is fixed to worlds up for better gameplay.
Min. 60 FPS and Framerate Independence	All time-based input/updates use deltaTime. Currently max FPS are not capped by the game and FPS are visible in the Games corner. Game can be played with constant 144FPS on FullHD.
Intuitive controls	<p>Input is implemented via polling or callback, depending on the type: WASD/Arrow Keys = Control Player Space = Start game: start movement and timer, pause game, restart game , = volume down . = volume up Esc = Close Window Enter = Fullscreen F1 = Unchanged H = toggle HUD during runtime</p> <p>Controls that will be disabled for the final game, but are implemented: 1 = Switch between Player- and DebugCam WASD = Control DebugCam M = Move DebugCam</p>
Collision Detection (Basic Physics)	<p>Each GameObject has a CollisionShape and Object which is automatically created with the Geometries Maxs. Terrain consists of two CollisionShapes, a boxShape and a TriangleMeshShape in the form of a cylinder, so that Collision is detected in the 'frame' of the hollow tube and not inside the cylinder. CollisionDetection between the Player and each GameObject</p> <p>Subtype: Goal = Win Ground = Crash (Lose) Collectable = Collected, deleted, count++ This feature is implemented with Bullet.</p>
(Moving) Objects	Objects are implemented as GameObjects(classes that inherit from 'GameObject'), each with its own

	<p>automatically created custom CollisionBody.</p> <p>The Player moves with a constant speed in the direction of its front-vector, it can be sped up by pressing the Boost-Button (B). The orientation is changed with KeyInput (WASD/ArrowKeys). The camera follows the Player with a fixed lookat. The PlayerModel was made via a locally run version of Hunyuian and a picture of Icarus. The glb was then added via our 'ModelLoader', with the help of assimp.</p> <p>The collectables are randomly generated and move constantly at a set amplitude and speed. The spacing is set at every new Game randomly, with hot spots where more rings appear. They can be collected by the Player and are deleted at collision. The Collectables gloom and their geometry is made with the custom made method 'createRingGeometry'.</p> <p>The goal is static, made and loaded the same way as the PlayerModel and their collision is the win condition.</p>
Adjustable Parameters	The initial Screen Resolution is set with the window.ini file. Fullscreen can be toggled via Enter.
Illumination Model	<p>one lightsource: direction light</p> <p>all Objects have a material or texture normals are calculated for each game object</p>
Win/Lose Condition	<p>Win = Reach Goal within Time</p> <p>Lose = Crash into Ground Time is up</p> <p>Bonus = Number of collectables</p>
3D Geometry	<p>The Player and Goal Models were made via a locally run version of Hunyuian and a picture of Icarus. The glb was then added via our 'ModelLoader', with the help of assimp. The ModelLoader uses a given glb and default Material to go through each mesh and create a Geometry with a fitting ObjectType.</p> <p>The Collectables are simple shapes created via the Geometry class. The sphere to reflect the cubemap is created by simple geometry as well. The particles from the particle system use the same geometry scaled differently.</p>
Bloom/Glow	The bloom effect is applied to each collectable using two framebuffers. The first contains the normal scene and the second contains only the objects lit with alpha = 1. A shader blurs the bright objects. After that the scenes are blended together

	creating the glow effect.
Heads-Up Display	<p>Current Player score, Timer and FPS are displayed using a Heads-Up Display. Also winning and losing are indicated with a banner. There is also a Tutorial display viewed at the start of the game and a countdown runs after starting it. Pause and information on how the game can be restarted is also viewed with the hud.</p> <p>We blend a 2D scene over the 3D scene in front of the current camera. To view letters and numbers we used a bitmap. The bitmap was uploaded using the library stb_image. For each symbol in the bitmap the uv coordinate is calculated and mapped to the corresponding char in a string given. The shader used only renders the alpha = 1 parts of the bitmap creating the blend in our scene.</p>
Playable	<p>The exe opens a window in which the game can be started via 'Space'. The Player movement and timer starts after the countdown 3-2-1. The User has to collect as many Rings as possible without crashing into the ground and the goal needs to be reached before the time runs out. Each possible ending shows its own end-screen and sound. The game can be restarted.</p>
Documentation	<p>This file, a well structured Git with Issues and object oriented code with comments.</p>
Environment Map	<p>Environment / Skybox implemented as a cubemap. It is reflected on the sphere placed inside the goal. The cubemap uses pictures loaded by the stb_image library. The pictures are mapped to the insides of a cube geometry and rendered first in the Scene Framebuffer Object.</p>
Vertex Shader Animation	<p>Water-vertexShader is added and normal Texture-fragShader is changed to work with it as well. VertexShader generates multiple waves and 'ripples' with given input data (amplitude, frequency,...) and combines them to one output = result_wave.</p> <p>WaterMaterial is a child of TextureMaterial and uses these Shaders. Water is then created as an obstacle.</p>
Particle System	<p>The CPU particle system simulates sparkles inside the goal coming from the reflection sphere using a pool of particles that are updated on the CPU. Each particle stores its own position, velocity, size, color and remaining life time. Dead particles are recycled using a respawn function that reinitializes their data. The particles are updated every frame,</p>

	<p>moving them and fading their alpha based on life time. After the update, position and color data are uploaded to the GPU into a dynamic buffer. Rendering is done via instanced drawing using a low-poly sphere and two per-instance attributes: position + size, and color.</p>
Textures	<p>Textures are generated in the Texture-class with dds-files which were generated from free online jpg-textures, LVA-webpage and local Hunyuan3D-version. Textures are afterwards used to create different kinds of Materials.</p>
Tessellation from Heightmap	<p>The Tessellation-Shaders are implemented and Terrain-class added. Tessellation part does not work atm because the given function Shader::loadShader throws errors at vertexShaderLoading, but a Heightmap is read with the Array2D-class and displayed as Terrain/Ground in-game.</p>
Music and Sounds	<p>Irrklang is used for in-game sounds and music. A sound-class was made and handles the different sounds that are made by winning, clashing, collecting and when running out of time. Once the game starts, music starts playing which volume can be controlled by pressing , for volume-down and . for volume up.</p>
Advanced Gameplay	<p>We have integrated music & sounds to allow the player to experience the game with different senses. There is also a Tutorial HUD explaining the user input. When the game is finished you can restart it by pressing space. You can also always pause the game during runtime, also with the space key. The spacing of the collectables is random for every game to create a new game experience every time.</p>