

Note: all online sources referenced in this document were last accessed on 10.06.2024

ROYAL HEIST		
GitHub	https://github.com/e1007167/ptvc24-RoyalHeist	
Students	Jan König, 01007167 Julia Weiss, 11908091	
Genre	Jumping Game	
Goal	Pick up all three treasures and then escape in time	
API	OpenGL	
Dev. Status	Finished	
Story	In the kingdom's heart gripped by tyranny, the resistance launches a fiery assault on the royal district. This is but a diversion concealing the master thief's infiltration of the castle. With this attempt to steal the royal artifacts of power, they seek to overthrow the spiteful ruler.	
Gameplay	Navigating a narrow path avoiding falling off and reaching the end of the level within the given time. Along the way the player must employ daring jumps and collect all treasures. Falling into the abyss or running out of time should be avoided to win.	
Controls	<ul style="list-style-type: none"> ● WASD: move player character (no diagonal move) ● Space: Jump ● Mouse: rotation and pitch ● ESC: Quit ● P: pause / continue ● N: quick restart (skips intro) ● M: mute / unmute ● U: unlimited time (only if presentationMode is active) ● ENTER: toggle Fullscreen ● F1: toggle wire-frame mode ● F2: toggle back-face culling mode ● F3: decrease gamma ● F4: increase gamma ● F5: toggle FPS counter 	
FEATURES		
optional Gameplay	Collision Detection	Player can collide with objects and floor
	Advanced Physx	Character controller with applied forces
	Advanced Gameplay	Collectables and a timer
	HUD	Displays the remaining time, text notifications
Effects	Modelling - CPU Particle System	Rising Ash and falling fire sparks as a global weather effect
	Animation - GPU Vertex Skinning	Player character animation
	Texturing - Dynamic Environment Map	An info point in the starting area implements a cubemap for dynamic environment mapping
	Post Processing - Lens Flares	Looking at the sun creates a lens flare effect

Royal Heist uses **OpenGL**, **PhysX** and **CMake**. Some of the basic mechanics include jumping or moving. **Advanced gameplay** elements include collectables, three endgame states as well as a timer. To win, pick up all collectables and reach the end of the level. To lose, run out of time or fall to your death.

A **camera** is implemented in the class `RoyalCamera.h`. The camera is based on a guide¹ but has been modified. Most noticeably, the camera is bound to a PhysX controller and cannot move individually. Instead, input moves the controller and the camera follows with an offset.

PhysX 5 is being used for various physics simulations:

- **collision detection** (e.g. the controller collides with objects)
- **raycasting** (s. Lens Flare for description of the implementation)
- **moving object**² (the player character representation in the PhysX engine)

Advanced Physics is implemented by using a kinematic object for the player character which is moved by applying forces via displacement vectors and gravitational pull. Furthermore, raycasting uses callbacks to the PhysX engine to react to certain events (e.g. the player model blocks the sun).

An **illumination model** provides a material to each geometry. **Materials** define the look of objects by providing an alpha value and attenuation. A **directional light** has been defined representing the sun. **Normal vectors** define how light interacts with geometries.

Textures are being loaded via **DevIL** and **STB**. For this to work an **image loader** class exists and multiple texture classes extending the template's texture class. Textures are either provided as JPG files or as DDS files with an alpha value and no pre-generated mipmaps. **Mipmaps** are generated by OpenGL via specification in the texture classes.

3D Models are being loaded via **Assimp**. Three collectables & the player model are loaded via the package `Model`. The implementation is based on the guide³ referenced on TUWEL.

A **HUD** (s. package `GUI` and `FreeType`) has been implemented using a 2D quad rendering **text via FreeType** (according to this guide⁴) and a 2D quad rendering a GUI-background. The HUD is used for showing a countdown and various text notifications during the game.

Variables can be changed via the settings file or live user input:

- automatic detection of the monitors aspect ratio can be toggled via the settings files variable `monitorAspectRatioDetection`
- window width & height can be adjusted in the settings file
NOTE: set `monitorAspectRatioDetection` to false to test custom window resolution
- vSync can be toggled in the settings file
- gamma can be set in the settings file (range: 0.01, 10.0) or via player input
- fullscreen mode can be toggled via the settings file or via player input

¹ LearnOpenGL Tutorial: <https://learnopengl.com/Getting-started/Camera>

² Controller: <https://nvidia-omniverse.github.io/PhysX/physx/5.2.1/docs/CharacterControllers.html>

³ LearnOpenGL Tutorial: <https://learnopengl.com/Model-Loading/Assimp>

⁴ LearnOpenGL Tutorial: <https://learnopengl.com/In-Practice/Text-Rendering>

GPU Vertex Skinning has been implemented based on a tutorial⁵ and a YouTube playlist⁶. The player character uses skeletal animation and GPU based skinning. A hierarchical bone structure for each animation (generated via Adobe Mixamo⁷) is loaded via Assimp. The current animation is dependent on game state & player input. Depending on the delta since the last render two neighboring key frames of the animation are chosen and their bone transforms are interpolated to create a framerate independent animation.

Furthermore, **Animation Blending** has been implemented by adding a second interpolation step. Two animations are tracked at all times, the new animation, and last animation. A blend factor is computed by normalizing the delta since the new animation started. For each bone of the new animation the corresponding bone of the last animation is fetched. For this pair of bones one can't simply interpolate between their local transforms. Instead, we interpolate their pairs of translation, rotation and scale matrices from which we can infer the final local transform.

A Lens Flare effect is implemented in the package Effects/LensFlare using 2D-quad rendering based on a tutorial by ThinMatrix⁸ and a guide⁹ referenced on TUWEL. When looking at the sun the effect is visible. The visibility is controlled in three ways:

1. the sun's position in clip space defines the brightness of the effect
2. the distance to the info point object affects the brightness of the effect
3. **raycasting** for toggling the effect when an object (e.g. jumping player) blocks the sun

A **CPU Particle System using instancing** was implemented. The effect is based on the guide¹⁰ referenced on TUWEL. The implementation can be found in the package Effects/ParticleSystem. The effect is used as a global weather and can be seen anywhere in the scene. Two different modes of operation exist:

1. spark mode: renders sparks of a fire falling down (reddish particles)
2. ash mode: renders ashes slowly rising up into the air (grayish particles)

A **cubemap and skybox for dynamic environment mapping** are implemented in the package Effects/CubeMap. The implementation is based on a guide¹¹ referenced on TUWEL & a guide by ThinMatrix¹². The skybox renders the horizon in the distance and the dynamic cubemap is used for the info point object at the very start of the level. Since the scene is black behind the starting point of the level, the effect is best seen when running around the info point and looking at it from the other side. The cubemap reflects the scene dynamically at 60 FPS. A virtual camera takes snapshots from the cubes POV for each face which are used for sampling.

irrKlang is used to play **sounds**. A custom sound manager class is being used (s. package Sound) which can fade sounds & effects conditionally using a complex logic provided by us.

⁵ LearnOpenGL Tutorial: <https://learnopengl.com/Guest-Articles/2020/Skeletal-Animation>

⁶ OGLDEV YouTube Tutorial: https://youtu.be/r6Yv_mh79PI?feature=shared

⁷ Adobe Mixamo: <https://www.mixamo.com/>

⁸ ThinMatrix Tutorial: <https://youtu.be/OiMRdkhvwqg?feature=shared>

⁹ Lens Flare Tutorial: <https://youtu.be/OiMRdkhvwqg?feature=shared>

¹⁰ OpenGL-Tutorial Tutorial:

<https://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/>

¹¹ LearnOpenGL Tutorial: <https://learnopengl.com/Advanced-OpenGL/Cubemaps>

¹² ThinMatrix Tutorial: https://youtu.be/IW_jqtJORc?feature=shared

Notable **additional features** that are not explicitly part of any effect or gameplay category:

- **blur** used for pause screen (s. package Effects/Blur and the blur shader package)
- complex logic for managing **animation blending** (s. GPU Vertex Skinning above)
- render **FPS counter** to screen (enable via settings.ini or by pressing F5)
- a **pause menu** using a **scene texture** (or render target) with post processing applied
- **game state based loading screens** (title, loading, failed, success) including complex logic for fade ins and fade outs (s. Package GameStateScreen and GameLogic)
- **game state based shader, sound and sound-effect modifications** (e.g. light color and alpha values change when getting closer to the info point, pitch and echo applied during pause)
- a **level builder** (or more accurately, a class to handle all scene objects) which creates PhysX objects, geometries or both depending on the parameters of a custom game object (s. package GameObject)
- a **quick restart function** (press 'N')
- **narration** for the intro and outro as well as sounds. All were manually edited to create a uniform soundscape
- **window aspect ratio automatically matches monitors** (unless deactivated in settings.ini)
- a **presentation mode** is implemented, which grants **unlimited time**
- the game can be **muted and unmuted** via keyboard callback

Guide

Before the time runs out:

1. collect all three treasures
2. only then reach the goal which is now highlighted by a compass

Sources & Libraries

- [PhysX 5](#) for collision detection and a kinematic player controller object
- [glad](#) (including [KHR](#)) for easier access of OpenGL functions
- [Freetype](#) for text rendering
- [DevIL](#) for loading DDS-image-files and [STB](#) for loading JPG-image-files
- [Assimp](#) for loading models
- [irrKlang](#) used as the audio engine
- textures: [level textures](#), [sun](#), [bag of gold](#), [chest](#), [map](#), [skybox](#), [player model](#)
- [animations](#), [compass](#)
- sounds: [fire](#), [bgm](#), [narrator](#), [battle sounds](#), [step](#), [thud](#), [jump](#), [item](#), [ambient drone](#), [game over](#), [item pickup](#), [win](#), [clock](#)