Gameplay

Playable (11 Points)

The alien represents the player.

The alien moves continually forwards.
The player can control the alien with the keys W, A, S, D and the arrows. The player can increase and decrease the speed of the alien with the W / upwards arrow (increase) and the S / downwards arrow (decrease) keys. With the keys A / leftwards arrow the alien will be moved to the left, with D / rightwards arrow the alien goes to the right side.
The speed of the alien can increase and decrease until a certain limit, the same applies to the left/right movements.
The goal of the alien is to reach the "gold" planet before the time expires and to avoid the obstacles in its way (asteroids & fire). The player has 1 minute and 10 seconds to finish the game and 3 lives at disposal.
In the game various 3D Objects can be found, all with different behaviours: hearts, coins, asteroids, fire and "smoke". Every time the alien collides with an asteroid, it loses a life and all the points it collected so far; if he has no life left, the game is over.
The player can collect lives on the way home. These are represented by hearts, and when the alien collects a heart, the number of lives increases by 1; this applies only if the number of lives is less than 3 and obviously more than 0;
If the player collects a coin, the number of points will increase and 3 seconds per coin will be added to the timer if the player didn't reach the finish line when the time has expired. If the player collides with an asteroid and consequently loses the collected coins, no extra time will be added at the end.
Every now and then the player encounters different types of smoke/fire cloud. Depending on the kind of smoke/fire, the alien picks up or loses speed. If the player chooses to surf through the mysterious smoke, the alien gets a speed boost that will last for 10 seconds, reaching a velocity that exceeds the one that the player would obtain by clicking the W / upwards arrow command, but the player's vision becomes impaired (it means that the brightness of the game will decrease for 10 seconds). This gives the player a risky but efficient way to make up for lost time. Surfing through the fire should be avoided since it will result in the alien losing speed and a life.

3D Geometry (6 Points)

In the game various 3D objects are present:

- alien
- asteroid
- heart
- coin
- flying disc/surfboard
- background objects (ancient stone, planets)

The asteroids, hearts and coins and surfboard are models taken from the internet. The applied textures were also taken from the internet.
The alien was built with blender (inspiration taken from different online tutorials).

The 3D objects are loaded with two classes: "Mesh", "Model".

"Open Asset Import Library" or Assimp is the library used for importing models:
http://assimp.org/index.php/downloads

In order to load 3D models this two tutorials were used:

- https://learnopengl.com/Model-Loading/Model
- https://learnopengl.com/Model-Loading/Mesh

All the objects are illuminated with a directional light.

Win/Loose Condition (3 Points)

The game has a countdown timer (in main class). This simple clock counts down from 70 seconds. "You Won!" will be displayed if the alien has reached the "gold" planet in time, otherwise the text "Game Over!" will be shown in addition to the reason why the player has lost: if time has run out the additional text will be "Time is up"; if the alien has no more lives at disposal, the text will be "You died!", otherwise if the player has reached the goal line but not the planet (in other words if the player is located next to the planet), the text will be "You missed the planet".

Intuitive Controls (2 Points)

Used keys: W / A/ S / D or arrows in order to move alien right or left and in order to increase/ decrease speed. Movements are multiplied with deltaTime in order to be framerate independent.

Intuitive Camera (2 Points)

One camera follows the player constantly and can rotate 360° in every direction.

Textures (2 Points)

Each object has at least at least one texture. Textures have mipmapping and trilinear filtering enabled.

Moving Objects (2 Points)

In the game the objects that move are the asteroids. These can go in different directions for example left and right or up and down.
The alien moves constantly forwards. The player can't stop the alien.
The movements are framerate independent.

Adjustable Parameters (1 Point)

All of the parameters can be adjusted in the settings file. Pressing the key 'F4' will show the current FPS.

Optional

Physics Engine (12 Points)

In our game the PhysX Engine is used for the collision detection. The alien is built as a character controller and the asteroids, coins, hearts, smoke/fire and planet are all PxRigidDynamic bodies.

Heads-Up Display (4 Points)

The HUD shows aspects of the game that are important for the player:

- countdown timer indicates how much time the player still has until the end of the game
- number of lives left represented by the image of an alien
- accumulated points
- in additional (for debugging purpose) fps.

The key to toggle the HUD is F3, for fps F4

In order to render the HUD, the class "Text" was implemented.

In order to load fonts and render them to bitmaps, the library "FreeType" was used: https://www.freetype.org/

For the rendering of the text the tutorial https://learnopengl.com/In-Practice/Text-Rendering was followed.

Effects

Lighting

- **Shadow maps with PCF (16 Points)**

  Shadow maps calculate shadows via rendering the scene from the view of a light source.

  The shadow map is implemented in the class "ShadowMap" and it follows the tutorial https://learnopengl.com/Advanced-Lighting/Shadows/Shadow-Mapping.

  The 3D Objects used for the game have a shadow (background objects are not included).

  The scene is rendered from the light's point of view, in this case a directional light, that is positioned in the upper/right side of the scene. Every object that can be seen from this point is lit; the others will be in shadow. In the game it can be seen, that the alien and the two asteroids project their shadows on the floor.

  In order to create the depth map, that will be used to calculate the shadow, and because the shadow map needs to be stored in a texture, the first step is to create a framebuffer. The framebuffer is created in the constructor of the ShadowMap class.

  Because the light source is a directional light (parallel rays) an orthographic projection matrix is used.

  The second step is to render the scene from the light point of view and to store the depth information in a texture. In order to do that two shaders (shadowMap.vert and shadowMap.frag) were implemented. The scene is rendered in the method drawShadowMap(). After that, a texture, that contains the information for calculating the shadow, is created.

  The third step is to render the "normal scene" and to add the shadows. This is done through

the textureShadow.vert and textureShadow.frag shaders. Here it will be check if the current fragment is in shadow or not.

Advanced Modelling

- **GPU Particle System using Compute Shader (12 Points)**

  A large amount of small, similar smoke particles is rendered. Each object has its individual data (position, velocity). Data is updated on GPU-side with a Compute Shader.

  The GPU particle system is implemented in the class “Particles” and “ParticleShader” and it follows the explanation illustrated in the PDFs “ComputeShader_SS18” and “GPU_Particles_SS18”. The render of the texture follows the tutorial “https://www.youtube.com/watch?v=POEzdiqEESo”.

  For this type of particle system 4 different shaders have been created:
  compute, vertex, geometry and fragment. The class “ParticleShader” is
  responsible for the creation and loading of the shaders”.

  In the implemented particle system, every particle has as attribute a position (vec4) and a velocity (vec4). In both cases the fourth coordinate is used in order to save the TTL (time to live) of the particle. The initial position of every particle is calculated randomly, but is based on a precise point in order to create a fire/smoke in this location of the scene. The life’s duration of a particle goes from 0 to TTL. TTL is a parameter, that is also calculated randomly and saved as the fourth coordinate of the velocity. The fourth coordinate of the position contains the actual life’s status of the particle and it starts from 0. When this value reaches the previously calculated TTL value the particle is regenerated.

  The update of the particle is calculated in the compute shader; here the position of every particle increases based on its velocity and when the particle dies (reaches the TTL value) the initial position is re-established and the life’s duration of the particle restarts from 0.

  In the class “Particles” the atomic counter and four SSBO are created, two for the updating of the position, and two for the velocity; these are used for the incoming and outgoing values.

  In order to animate the texture of the particles, a texture atlas was used.
  Based on how long the particle was alive a different stage of the texture to render is chosen. This operation is computed by changing the texture coordinates. To calculate these coordinates it is necessary to define the number of rows in the texture. Then, to determinate where on the texture the top left corner of the part that should be used is, two offsets are calculated in the vertex shader. The second offset is used in order to indicate where the next stage is. A blend value is also computed, which establishes how much the current texture stage should be faded into the next texture stage.

  In the geometry shader every particle, represented by a single vertex, is transformed into a quad with four vertices and the texture coordinates, that indicate which portion of the texture will be render, are calculated.

  The geometry shader passes than to the fragment shader the coordinates that represent the two stages of the texture and the blend factor. The final colour is determined by mixing these two colours together using the blend value.

Animation

- **Hierarchical Animation (4 Points)**

  A flying disc/surfboard has been placed under the alien. This disc spins around its own y-Axis and follows constantly the alien everywhere it goes. In order to achieve this animation a hierarchical animation was implemented, where the alien represents the "parent" of the hierarchy and the disc the child. For this reason, every time the disc is rendered, in addition to the calculation of the rotation, the model matrix of the disc is also multiplied with the model matrix of the parent.

  For this part the used code was taken from exercise 6 of ECG.

Shading

- **Cel Shading (4 Points)**

  Conventional (smooth) lighting values are calculated for each pixel and then mapped to a small number of discrete shades to create a characteristic flat look. This operation is implemented in the shader responsible for the rendering of the 3D objects "textureShadow.frag". There are 3 different level of shades and based on the original colour of the fragment, the colour is approximated to one of these discrete values.

Post Processing

- **Bloom/Glow (8 Points)**

  The bloom effect is implemented across the classes Main.cpp and BloomShader.cpp as well as the vertex and fragment shaders bloom, blur and bloom_final. The implementation follows the tutorial https://learnopengl.com/Advanced-Lighting/Bloom.

  Currently the implementation does not work correctly, which is why the respective code has been commented out in the main class. I am still working on fixing the issue and would be more than happy about the opportunity to present the working version at the game event.