

The Jupiter Event

Sebastian Benjamin Apfelthaler (00226768)

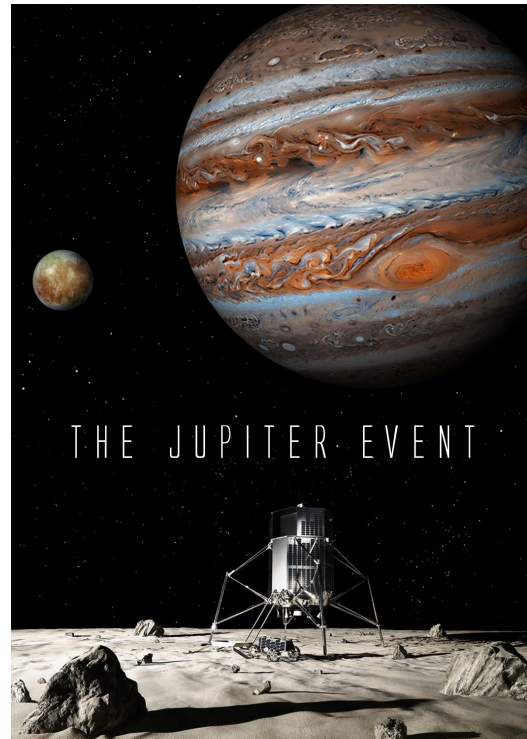
Alexander Cech (08900070)

GitHub: [cgue19-TheJupiterEvent](#)

Setting und Hintergrundgeschichte

Die Monde Jupiters wurden von einem Mega-Konzern erschlossen. Mittels autonomen Sammelfahrzeugen wird auf den Mondoberflächen Helium-3 gewonnen, das sowohl auf der Erde, als auch in Raumschiffantrieben, zur Kernfusion benötigt wird.

Wartungsmitarbeiter des Konzerns befinden sich in einer Umlaufbahn, als der Jupiter plötzlich einen starken Strahlungsimpuls abgibt. Dies beschädigt den Tank ihres Raumschiffes, der daraufhin nur noch eine kleine Menge Treibstoff fassen kann. Des Weiteren spielen die autonomen Sammelfahrzeuge durch den Strahlungsimpuls verrückt und sind stehen geblieben oder fahren sinnlos umher.



Spielprinzip

Der Spieler kontrolliert ein raketenähnliches Raumschiff in 3rd-Person-View in der Nähe einer Mondoberfläche. Steuerdüsen ermöglichen die Drehung um die Hauptachsen. Ein Haupt-Thruster beschleunigt das Schiff in Richtung der aktuellen Orientierung.

Der Treibstoff des Haupt-Thrusters ist stark begrenzt und muss oft nachgetankt werden. Dies geschieht durch Landung auf Helium-3-Sammler-Fahrzeugen, die das Schiff sowohl mit Flüssigtreibstoff, als auch mit Helium-3 betanken. Flüssigtreibstoff wird benötigt um sich auf dem Mond fortzubewegen. Der steigende Helium-3-Tankstand repräsentiert den Level-Fortschritt. Hat man genug Helium gesammelt, entkoppelt sich der Oberteil des Schiffes. Nun gilt es eine gewisse Mindesthöhe zu erreichen um ins nächste Level fortzuschreiten.

Die Herausforderung des Spiels besteht darin, das Schiff mit begrenztem Tank sicher zum nächsten Sammelfahrzeug zu navigieren und sicher zu landen. Kommt man zu schnell oder zu schräg auf, so explodiert das Schiff und man hat verloren.

Das erkundbare Spielfeld ist durch Strahlungszonen an den Levelgrenzen und überall ab einer gewissen Höhe begrenzt. Je nach Level erhöht sich die Schwierigkeit durch noch in Fahrt befindliche Sammelfahrzeuge, durch stehende Fahrzeuge an Stellen, an denen eine Landung durch das gegebene Terrain schwierig ist und durch Strahlungszonen innerhalb

des Levels. Die nötige Planung, welche Sammelfahrzeuge man als nächstes ansteuern möchte, erhöht die taktische Tiefe. Des Weiteren unterscheiden sich die Level durch unterschiedlich starke Gravitation.

Das Level ist geschafft, sobald genug Helium-3 gewonnen wurde und man danach in große Höhe vordringt. Gewonnen hat man durch Bewältigung des letzten Levels.

Controls

Das Schiff wird durch Aktivierung seiner vier Steuerdüsen-Arrays (Tasten W,S,A,D,Q,E) frei um seine 3 Raumachsen gedreht und durch Zündung des Hauptantriebs (SPACE) beschleunigt. Die Kamera kann mit der Maus frei um das Schiff gedreht werden (Arc-Ball-Kamera) und mittels Mausrad weiter entfernt bzw. näher herangebracht werden.

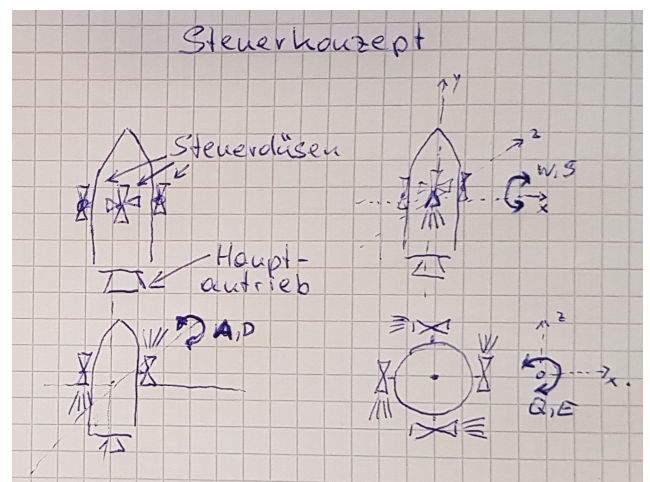
W,S	Rotation um die lokale X-Achse
A,D	Rotation um die lokale Z-Achse
Q,E	Rotation um die lokale Y-Achse (Längsachse)
SPACE	Hauptantrieb

Flugassistent (Beschreibung siehe weiter unten)

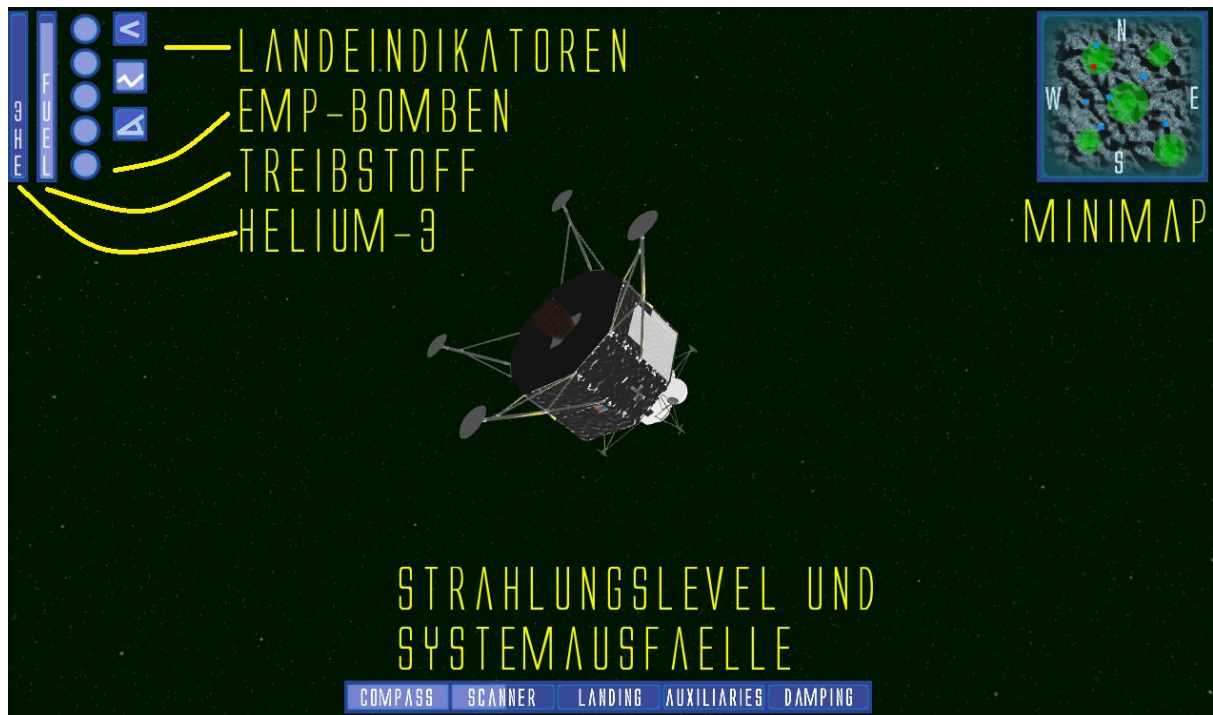
linke Maustaste	C	Prograde
rechte Maustaste	Y	Retrograde
mittlere Maustaste	X	Normal stellen

Weitere wichtige Features (für eine Komplettübersicht: siehe technischer Bericht)

B	EMP-Bomben-Abwurf
P	Pause
R	Reset (vom letzten Speicherpunkt neu starten)
Shift R	Level komplett neu starten
Alt Return	Vollbildmodus umschalten



HUD



Gameplay

Flug

Das Spiel simuliert weitgehend ein realistisches Flugverhalten im Vakuum nahe der Oberfläche eines Mondes mit realistischer Gravitation. Aufgrund der fehlenden Atmosphäre wird ein bestehender Flugvektor nur von der Gravitation und von Steuerungsbefehlen beeinflusst. Zur leichteren Orientierung stehen sowohl eine Minimap, als auch ein Kompass als HUD-Elemente zur Verfügung.

Bei Kursänderungen ist der bestehende Flugvektor zu berücksichtigen. Fliegt man beispielsweise Richtung Norden, möchte aber Richtung Nordosten fliegen, so möchte man nicht einfach direkt nach NO beschleunigen. Stattdessen sollte man sein Schiff Richtung Osten drehen und in diese Richtung beschleunigen um den Flugvektor insgesamt Richtung NO zu verändern. Dieses Vorgehen spart Treibstoff durch weniger erforderliche Bremsmanöver und senkt das Risiko von Kollisionen mit der Umgebung durch zu hohe und nicht mehr rechtzeitig zu bremsende Geschwindigkeiten.

Um gegen die aktuelle Flugrichtung zu bremsen, ist es erforderlich, das Schiff entgegen die aktuelle Flugrichtung zu drehen um dann zu beschleunigen (zu "bremsen").

Flugassistent

Angelehnt an real existierende Flugassistenten für bemannte Missionen wurden in unser Spiel drei sehr hilfreiche Modi implementiert, die durch die Maustasten aktiviert werden können.

Prograde: Das Schiff orientiert sich in Richtung des Flugvektors und lässt diesen somit erkennen. Hilfreich z.B. bei obigem Beispiel der Kurskorrektur nach NO. Nach Schubmanöver nach Osten kann man mittels "Prograde" die Orientierung des resultierenden Flugvektors einnehmen. Auch geeignet, um weiter in Flugrichtung zu beschleunigen. Des Weiteren dreht sich das Schiff um die eigene Achse in stets die gleiche Position um die Steuerung zu erleichtern.

Retrograde: Das Schiff orientiert sich entgegen des Flugvektors. Die Hauptanwendung stellen Bremsmanöver dar. Dies können "Stops" in größerer Höhe sein um sich anschließend manuell zu drehen und in eine neue Richtung zu fliegen, oder Notfallsbremsungen, wenn man in das Terrain zu "krachen" droht. Des Weiteren stellt es ein wichtiges Hilfsmittel bei Landungen dar.

Normal: Das Schiff richtet sich normal zur ebenen Oberfläche aus. Wichtig bei Landungen, oder um im Notfall schnell an Höhe gewinnen zu können. Auch nützlich, wenn der Kurs passend ist und man nur die Gravitation ausgleichen möchte.

Landung auf der Oberfläche

Es empfiehlt sich, mittels "Retrograde" das Schiff in Bodennähe noch "in der Luft" zum Stopp zu bringen und - sobald man kaum noch eine Bewegung ausführt - "normal" zu aktivieren. Nun gleitet man durch die Gravitation und richtig orientiert nach unten und kann mittels "Hauptschub" den Fall abbremsen.

Das HUD zeigt drei hilfreiche Indikatoren. Für eine sichere Landung müssen alle drei leuchten. Leuchtet auch nur eines nicht, so resultiert die Landung in einer spielbeendenden Explosion.

Die Indikatoren:



"Landing Speed" - man darf nicht zu schnell aufsetzen

"Surface Angle" - das Terrain darf nicht zu steil sein

"Relative Angle" - das Schiff darf nicht zu schräg zum Terrain stehen

Fahrende Sammler und EMP-Bomben

Auf Helium-3-Sammlern muss gelandet werden um Treibstoff aufzutanken und um Helium-3 abzapfen. Hat man genügend Sammler belandet und somit genügend Helium-3 gewonnen, so ist das Level fast (siehe weiter unten) geschafft. Sammler sind auf der Minimap eingezeichnet und somit in Kombination mit dem Kompass zu suchen.

Während das Landen auf stehenden oder sehr langsam fahrenden Sammlern möglich ist, so ist dies auf schnelleren Sammlern zwar theoretisch möglich, praktisch jedoch sehr schwierig. Bei den fahrenden Sammlern ist es daher erforderlich sie vor einer Landung auf ihnen zu stoppen. Dies geschieht durch Abschuss von EMP-Bomben. Diese senden einen elektromagnetischen Impuls aus, der das Fahrzeug stoppt. Zu beachten hierbei ist der Detonationsradius von grob der fünffachen Länge eines Sammlers, sowie die zeitliche Detonationsverzögerung von einigen Sekunden. Gutes Timing des Abschusses und Miteinkalkulierung des gegenwärtigen Flugvektors und der Flughöhe sind daher erforderlich. Zu beachten ist die Limitierung auf fünf Bomben, die jedoch wieder aufgefüllt werden können (siehe Strahlungszonen). Auch muss eine gewisse Mindestgeschwindigkeit relativ zum Boden erreicht sein um Bomben abwerfen zu können.

Achtung! Nach 90 Sekunden nehmen gestoppte Sammler wieder die Fahrt auf! Eile bei der Landung ist also geboten. EMP-Detonationen führen auch zu Ausfällen auf dem eigenen Schiff, falls die Bomben zu nahe detonieren!

Die Bomben treten aus der "Bauchseite" des Schiffes aus. Es empfiehlt sich, "Prograde" oder "Retrograde" zu aktivieren, was den Bauch Richtung Boden zeigen lässt. Bomben können am Terrain abprallen und entlang rollen, sofern sie den Boden erreichen bevor sie detonieren - dies lässt sich taktisch nutzen.

Landeroutine für stehende Sammler

Steht der Sammler, so empfiehlt sich für Anfänger folgende Landeroutine:

- Landung wie oben beschrieben auf dem Terrain - knapp neben dem Sammler
- ein kurzer "Hüpfer" mittels Hauptschub währenddessen man das Schiff um die eigene Achse so ausrichtet, dass pressen von "W" (Vorwärtsskippen) das Schiff Richtung Sammler neigen lassen würde (aber "W" noch nicht ausführen!)
- wieder aufsetzen nach dem "Hüpfer"
- nun mittels Hauptschub leicht abheben, vorwärts kippen und nochmals vorsichtig Schub geben
- normal stellen mittels Flugassistent
- nun gleitet man aufgerichtet auf den Sammler zu und muss nur noch mittels Hauptschub die Höhe anpassen, bis man auf der Landefläche aufkommt

Bei langsam fahrenden Sammlern kann man, um eine Bombe einzusparen, entlang der Sammlerbahn landen und auf den Sammler warten und auf ihn mit obiger Methode "aufspringen". Erfahrene Piloten können jedoch auch direkte Landeversuche auf langsam fahrende Sammler wagen und sich dadurch eine Bombe und den Umweg über Terrainlandungen ersparen.

Strahlungszonen

Strahlungszonen an den Levelkanten und überall ab einer gewissen Höhe dienen der Spielfeldbegrenzung. Ein Flug hinein bewirkt einen Fade-to-Red-Effekt, sowie steigende Strahlungsbelastung für das Schiff. Akustisch macht sich Strahlung durch einen Geigerzähler-Ton bemerkbar, der je nach Strahlungslevel variiert.

Des Weiteren gibt es innerhalb des Spielfeldes zylinderförmige Strahlungszonen. Diese erfüllen den Zweck, eine zusätzliche Herausforderung zu schaffen und um darin den Füllstand von EMP-Bomben wieder aufzufüllen. Diese Bereiche werden durch ein Fade-to-Green bemerkbar gemacht, und sind als grüne Kreise auf der Minimap markiert. Zu beachten hierbei ist, dass die Strahlung Richtung Zentrum zunimmt und eine Mindeststrahlungsintensität (aktuelle Verstrahlungsrate, nicht die kumulierte Strahlung) vonnöten ist, um den Bombenaufladevorgang in Gang zu setzen. Achtung: Nur in grünen Strahlungszonen können Bomben aufgeladen werden, nicht jedoch in Strahlungszonen die der Levelbegrenzung dienen (rote Zonen).

Ausfälle

Je tiefer man sich in einer Strahlungszone befindet, desto schneller steigt die Gesamtverstrahlung des Schiffes. Am unteren Bildende befindet sich ein Balken der gleichzeitig Strahlenbelastung und die Funktionstüchtigkeit der Bordsysteme anzeigt. Der Strahlungsbalken wächst dabei von links nach rechts an und immer mehr Systeme fallen in der vorgegebenen Reihenfolge aus:

Kompass → Minimap → Lande-Indikatoren → Flugassistent → automatische Dämpfung

(Die automatische Dämpfung bewirkt den langsamen Stop von Drehungen des Schiffs um die eigenen Achsen)

Verlässt man die Strahlungszonen, so sinkt die kumulierte Strahlungsbelastung wieder. Auch selbst abgeschossene EMP-Bomben senden Strahlung aus, sobald sie detonieren. Je näher man sich der Detonation befindet, desto mehr Strahlung erreicht das eigene Schiff. Dementsprechend stark wächst auch der Ausfallsbalken.

Levelende, Fuel und Checkpoints

Pro Level befinden sich sechs Sammler, die man anfliegen kann. Jedoch entkoppelt sich bereits nach fünf komplett entleerten Sammlern die obere Schiffseinheit und ein Helium-Antrieb wird automatisch gezündet (dies wurde so designed um pro Level einen Sammler, der möglicherweise zu schwierig ist, auslassen zu können). Nun muss die Aufstiegseinheit wieder in den Weltraum gebracht werden, um das Level abzuschließen - dazu muss die obere "Strahlungsdecke" durchquert werden.

Nach dem Tankvorgang von Helium-3 auf einem Sammler wird ein Checkpoint aktiviert. Durch drücken von "R" kann man an dieser Stelle wieder starten und muss daher nicht das Level ganz von vorne beginnen.

Während jeder Sammler nur eine begrenzte Menge Helium mit sich führt, so kann konventioneller Treibstoff beliebig oft nachgetankt werden.

Empfehlungen für das erste Level

Der erste Sammler steht direkt unter dem Startpunkt des Spielers - für eine Landung sind hier keinerlei Steuermanöver notwendig, lediglich die Betätigung des Hauptschubs, um nicht zu schnell zu werden. Außerdem soll dieser Sammler eventgesteuertes Verhalten demonstrieren - er setzt sich in Bewegung sobald man landet. In unmittelbarer Nähe befindet sich ein weiterer Sammler mit sehr langsamer Geschwindigkeit und geradlinigem Kurs. Hier kann das Bombardieren mit EMP-Bomben geübt werden.

Weitere Gefahren

- In Level 2 und 3 wurde das Terrain um Kristall-Objekte erweitert. Es besteht Kollisionsgefahr!
- Im Level "Europa" gibt es Seen aus Wasser und auf "Io" Lavaseen. Eine Landung darin ist nicht empfehlenswert. Manche Sammlerrouten führen die Sammler teilweise unter Wasser. Hier sollte abgewartet werden, bis diese wieder auftauchen. Von Haus aus stehende Sammler können sich in Fahrt setzen, sobald man darauf gelandet ist - rechtzeitiges Abheben kann erforderlich sein um nicht ins Wasser mitgenommen zu werden.
- Manche Sammlerrouten führen durch verstrahlte Gebiete.
- Ab Level 2 ist ein starkes Brems- beziehungsweise Ausweichmanöver bei Levelstart vonnöten

Technischer Bericht / Technical Report

Da unser Code durchgehend englische Begriffe verwendet, ist der technische Bericht ebenfalls in englisch gehalten.

Basic data structures / classes

Geometry:

Contains vertex, normal and UV arrays, as well as generates VAOs for rendering.

Mesh:

Responsible for rendering a single geometry object; also holds material properties relevant for lighting as well as textures.

Model:

An encapsulation of one or more meshes – typically one in-game-object. Also stores the current world-transform of the object, and (if available) it's PhysX actor. Includes the facility to load objects from disk via the Assimp library (only to memory, all OpenGL stuff is handled by our code).

Player:

Representation of the player's rocket ship. Contains a model, physical properties, player status information, such as fuel and helium. Player physics are simulated via a Runge-Kutta-4 integrator. Auxiliary systems, like physically correct orientation changes by automated thruster actuation, are also handled in this class.

Crawler:

Represents one of the crawlers ("Sammler"). Holds multiple (currently 2) LOD-versions of the model, speed and status (remaining fuel, helium). Has a waypoint system to smoothly drive along a predesigned course, and implements an automatic leveling to the terrain, including tilting. Additionally a PhysX-actor is used for collision detection and to allow bombs to bounce off.

Bomb:

The encapsulation for an EMP-bomb. Apart from the model, holds a PhysX actor for bouncing around and a countdown timer (part of the model is shaded differently, depending on the time left to explode).

Heightfield:

This class defines the terrain; internally it uses a 16-bit grayscale heightmap texture, a normal texture (used for determining valid landing positions and orientations), a diffuse texture map, and a precalculated ("baked") light map that considers self-shadows. For rendering a tessellation control and evaluation shader is used to produce triangles that are approximately equally large on screen (adaptive tessellation). The diffuse texture is combined with the light map in the fragment shader.

Level:

Manages the loading of a level and holds all level-relevant information.

Game:

This is the main class, containing the render- and update-loop as well as keyboard and mouse control.

Shader:

A class to encapsulate a GLSL program; handles the loading and compiling of shaders from disk (including a hand-crafted `#include` extension), shader binding and the setting of variants.

Texture:

Encapsulates textures (2D generic, cubemaps, 16-bit grayscale); handles loading from disk and optional Y-flipping via the Devil library (only to memory; OpenGL textures are then created by our code)

Quad2D:

A helper class to render quads to 2D; used for HUD elements and overlays.

Camera:

This handles the arc-ball camera, which stays focused on the player.

Compass:

This class is responsible for drawing an overlay-compass, dependent on the current view direction.

ControlPanelNew, Minimap:

Draw the HUD elements (status indicators and the minimap).

Crystal:

Wrapper class for crystal objects, allowing to create them at designed positions, orientations and scale; also generates a PhysX model for collision.

Framebuffer:

Utility class to manage creation and switching of framebuffers.

Light, Material:

Classes and structs to represent lights and material properties. The current implementation

Physics:

Handles initialization and updating of the PhysX engine and contains some utility functions to convert PhysX-structures to glm and vice versa.

KDTree:

Class that handles the creation and update of a kd-tree for view frustum culling.

LensFlare:

Handles the lens flare effect.

Liquid:

Wrapper class for liquid surfaces, like water and lava.

Menu:

In game menu for level selection.

Particles:

Implements a GPU particle system, based on transform feedback. Used for the main thruster of the player objects (slightly different depending on whether the player is “separated” or not).

Sound:

Manages loading and playing background music, sound effects and voice. Based on the FMOD core sound system.

TextOutput:

Class to render arbitrary text to screen, based on the FreeType library.

TimedText:

Manages timed text overlays, i.e. text overlays visible for a specified time.

Brief description of some of the more interesting aspects

Modelling, audio, graphics tools

For 3D modelling we used Blender; to work with 2D graphics, paint.net and Gimp, and for audio editing Audacity were used. Terrain design was done with L3DT (see next Section).

Heightfield

An external tool (L3DT) was used to design height fields and generate texture maps as well as to pre-bake the sunlight illumination including self-shadowing of the terrain to a separate light map. Normal maps, which are required for our game logic, are created on the fly from the height map data during program execution.

Internally the terrain is drawn as a checkerboard-pattern of larger, flat, patches. A tessellation control shader (TCS), based on an example from the OpenGL SuperBible, adds the Y coordinates from the heightmap and then projects the patch corners to NDC space and determines the patch border lengths there. Depending on these lengths a tessellation level is chosen. At this stage, also view frustum culling for the terrain is performed, based on the normalized device coordinates: if all four corners of the patch fall outside the visible area, the patch is discarded. The tessellation evaluation shader then calculates the vertex and (diffuse) texture coordinates from the subdivided patch. The fragment shader then combines the corresponding texels from the diffuse terrain texture with the precalculated light map, and also calculates a mock-up-shadow of the player by comparing the vectors from the current terrain location to the sun and to the player respectively.

In addition a separate terrain model, based on the same heightmap, is built for PhysX collision detection as a PxHeightfield geometry.

Player physics and auxiliary systems

As we had not reached a definite decision about whether to use PhysX or not in the beginning, we implemented the player physics simulation manually, based on the online article series "Game Physics". The simulation uses momentum and angular momentum, as well as forces and torques, to integrate velocity, angular velocity (and finally position and orientation). As integrator a Runge-Kutta-4 (RK4) model is used.

Since we wanted the player's ship to behave as physically accurate as possible, the auxiliary systems are not allowed to just modify the ship's orientation directly; instead they have to actuate the physical control thrusters, just like the player can do, to eventually reach the target orientation.

Landing on the terrain surface or on a crawler was quite challenging. We finally built two (PhysX geometry) hitboxes around the player, one very flat just covering the landing struts ("landing-box"), and one covering the rest ("body-box"). Landing on terrain works like this: If a collision is detected between the terrain and the landing-box (but none with the body-box), and the ship's momentum is not too high, the terrain not too steep and the relative angle between the ship and the terrain's normal is small enough, landing should succeed. However we want the ship to stand level on the landed spot, i.e. point into the same direction as the terrain normal does. Just rotating the ship to its final orientation may result of a

collision of the body-box with terrain, or in loss of contact of the landing box with the terrain. If either of this happens, we internally push the (rotated) player gradually upwards/downwards until landing conditions are reestablished. Landing on a crawler works similarly, but in addition a check is made if the ship is inside the designated landing zone.

Lighting and texturing

A single directional light source - the sun - is used for Phong-shading the objects (player's ship, crawlers, bombs). The terrain itself uses a pre-baked light map which is combined with it's diffuse texture in the shader, and dimmed by the player's mockup-shadow.

The ship model is fully textured, the crawler and EMP bomb models are partially textured and partially colored (with according material properties). The bomb's colored part is faded from green over yellow to red to indicate the detonation countdown. Normal mapping is used on parts of the player's ship and on water surfaces (see Section Normal mapping).

Adjustments via settings.ini

Window-dimensions, as well as full-screen-mode, resolution, refresh rate and gamma-correction can be set via the settings.ini file in the assets folder. Additionally the background music can be enabled/disabled there, along with its volume.

Skybox

For the starfield background, including the greenish nebulae, a cubemap texture is used as skybox. Originally we had the sun and Jupiter painted into the skybox and rotated the box to match the sun position with the lighting conditions present in the current level. We later aborted this approach, since it was too inflexible and did not work too well with the lens flare effect, and now instead draw the sun (a sprite) and Jupiter (a lit, textured sphere) separately. The basic skybox was generated by an online skybox generator (see references).

Shockwave effect

For the detonation of the EMP-bombs we wanted a kind of shockwave or pressure wave effect. The current implementation is loosely based on an online example (see references) and uses distortion of the texture coordinates along an expanding radius. For this, the entire scene is rendered to an FBO first, and then the distorted version is rendered to screen as a quad.

Using glDrawArraysInstanced

Wherever possible, we avoided to pass simple, static vertex lists to shaders and instead used the vertex shader to "generate" the vertices by using glDrawArraysInstanced. This technique is used for all quad-rendering (overlays, HUD, minimap, post-shockwave, ...), as well as for the patches of the terrain. The vertex shader itself generates the vertex coordinates (and texture coordinates) from gl_InstanceID and gl_VertexID.

LOD switching

Level-of-detail switching is implemented for crawlers and crystals. For both, we use 3 different levels (high-poly, low-poly, very-low-poly), and decide which one to render by the current distance of the object to the camera.

Normal mapping

Although only a simple version of normal mapping was required, this was not sufficient for our purposes, so we implemented **full normal mapping** (including calculating tangents and transforming the light calculations to tangent space). This full version is used for rendering the metal foil wrapping of the player's ship, its top surface metal grid, as well as the two main thruster cones. A **simple** version of normal mapping is used on water surfaces (Level 2), where it is combined with a time-driven displacement map. Normal mapping on the player model can be turned on/off by pressing Ctrl-N.

Lens flares

The lens flares effect is based on the combined information from several online texts (see references). Basically it works by overlaying several translucent sprites to the main scene. At runtime, several textures of different shapes (spikes, halos, rings, blooms, ...) are created algorithmically. For performance reasons, a single "atlas" texture, containing all the types, is created, rather than a separate texture for each type. To render the effect, the sun position on screen is determined, and a set of 16 of those textures are drawn along the sun/screen center axis. Positions, size and colour intensity of the sprites depend on the distance from the sun to the center. Separate spikes are drawn with different color channels to mimic anisotropic lens distortion. The final choice of which textures to render at what distance, as well as their relative size and intensity was based on lots of experiments, settling for the most pleasing results.

Particle system (GPU, transform feedback)

Several online sources were perused to learn the basics of transform-feedback based particle systems (see references). Basically the system works by using two buffers, reading particle state from one, and writing to the other via GPU shaders; the role of the two buffers is then swapped for the next frame. Transform feedback is used to record the number of particles written to the buffer in the update pass, and to render that exact number in the render pass. Our particle state consist of type, time-to-live, position, velocity and color per particle. Both the update and render shader use a geometry shader for the update logic and to convert points to quads in the render pass. There is some randomness involved in the initial velocity of new particles; as pseudo-random source a 1D random texture is created at runtime and passed to the update shader.

Initially we had lots of trouble getting the transform feedback system to work correctly; most available tutorials did not use VAOs, but this is enforced by the requested OpenGL core version we use. Once these problems were sorted out, we experienced seemingly random jittering in the particle stream. Days of debugging the shader and code did not lead to any solution - until finally we found, that timing synchronization is crucial; since we only update

the player physics every 1/60th of a second, the particle stream update had to be synched not only to that same rate, but also to the exact same frame. Doing that finally resolved all visual glitches.

View-Frustum Culling, kd-Tree

View frustum culling on the terrain is built into its tessellation shader (see Section *Heightfield* above). For all other static and dynamic objects, we implemented a kd-tree to achieve performant culling queries. Different methods of partitioning the tree cells were tried and can be selected by changing the KDTree's class `m_splitStrategy` member. Dynamic objects are removed and re-added to the tree each time their position changes; both bounding spheres and (axis-aligned) bounding boxes are used: boxes for large flat objects, like the surface of lava and water, and spheres for the remaining objects. In order to determine if an object is to be rendered, in a first stage culling is performed recursively on the tree's cells. Only objects that contained in partially or fully visible cells are considered further. For objects in partially visible cells we perform an additional per-object visibility test, since tests have shown that the performance gained by this step slightly outweighs the additional costs for the tests. When developer-keys are enabled (see keyboard reference chart), pressing K prints the current number of culled objects to the console.

Crawler positioning

To position crawlers naturally on the terrain surface (especially when moving), we had to experiment for a long time to find visually pleasing and non-jittering results. The way it is done now, is to flatly project the crawler's 2D position onto the 3D terrain, obtain the terrain height for the two front and two back wheels, and interpolate those to determine the final center point height. Finally the 3D positions of the wheels and the forward direction are used to calculate an overall orientation quaternion. The final effect is that the crawlers follow the terrain naturally, rolling and tilting as required.

Variable screen/window size and full-screen switching

In order to implement in-game full-screen toggling, we needed to be able to deal with changing window sizes anyway, so we made the render window resizable. As a side effect, this proved very useful to test resolution independence of various parts, especially HUD and text overlays. To make this work, we implemented a callback system - every class/object that depends on screen size (or has made pre-calculations based on it), registers for a callback that is triggered whenever the size changes. Once this was working, toggling between full-screen and windowed mode was easy (key combination Alt-Return).

Gameplay and effects checklist

Gameplay-Compulsory: (complete):

Playable, 3D Geometry, Win/Lose-Condition, Intuitive Controls, Intuitive Camera, Textures, Moving Objects, Documentation, Adjustable Parameters

Gameplay-Optional:

Physics Engine, View-Frustum Culling, Heads-Up-Display

Effects:

Lighting: Lightmaps using separate textures

Terrain uses a diffuse texture plus precalculated light-map, combined at runtime.

Advanced Modelling: GPU Particle System using Transform Feedback

Used (differently) for the main thruster of the unseparated and separated player model.

Terrain: Terrain-tessellation from heightmap:

Uses a tessellation control and evaluation shader to subdivide terrain patches.

Subdivision is adaptive, according to the final patch size in NDC coordinates.

Animation: Hierarchical Animation:

The three wheel pairs of each crawler model are rotating when the crawler is moving, rotation speed matched to its current velocity.

Texturing: Procedural textures:

Procedural textures are used to create moving lava surfaces (Level 3). The shader is based on an existing example (see references) and modified for our purposes.

To some extent the water surfaces also include procedural components (time-driven distortion map combined with normal mapping to produce moving specular highlights).

Shading: Simple normal mapping

Used for several parts of the player's ship model (full normal mapping), as well as on water surfaces (simple normal mapping).

Advanced Data Structures: kd-tree

A kd-tree is used to determine view frustum culling for moving and stationary objects, using both bounding spheres and boxes.

Post processing: Lens Flares

Lens flares are rendered when looking directly into the sun.

Additional effects:

Skybox

Shockwave (at bomb explosion)

Sprite animation

The animated player explosion uses a sequence of 64 images; the images are combined into one texture; depending on the animation frame to display, different UV coordinates are chosen.

Fading effects

Fading of the main scene (when inside radiation zones) is created by overlaying a screen-sized quad with variable alpha channel.

References - external assets and libraries used

Libraries:

Sound system (FMOD core)	https://www.fmod.com/
Physics engine (PhysX)	https://developer.nvidia.com/physx-sdk
Image loading (DevIL)	http://openil.sourceforge.net/
Object loading (assimp)	http://www.assimp.org/
Text formatting (boost format library)	https://www.boost.org/
OpenGL framework (GLFW)	https://www.glfw.org/
OpenGL Mathematics (glm)	https://glm.g-truc.net/0.9.9/index.html
FreeType	https://www.freetype.org/

Tools:

Skybox generator	http://www.tyro.github.io/
Explosion Texture Generator	https://www.saschawillems.de/creations/explosion-texture-generator/
Terrain Editor (L3DT Pro)	http://www.bundysoft.com/L3DT/
Text-to-speech generators	https://www.naturalreaders.com/online/ https://ttsreader.com/

Materials:

Model: Mars Explorer (orig.)	https://free3d.com/3d-model/mars-explorer-82624.html
Model: Lander (original)	https://www.turbosquid.com/3d-models/lunar-surface-access-3d-lwo/856669
Model: Plasma grenade	https://free3d.com/3d-model/plasma-grenade-42384.html
Model: Crystal	https://free3d.com/3d-model/3d-crystal-29456.html
Jupiter sphere texture	https://www.turbosquid.com/3d-models/planet-jupiter-3d-model/1056891
Image: Jupiter	https://www.mnn.com/earth-matters/space/blogs/jupiters-famous-red-spot-insanely-hot-scientists-probe-explanations
Moon images	https://en.wikipedia.org/wiki/Ganymede_(moon)#/media/File:Ganymede_g1_true-edit1.jpg https://en.wikipedia.org/wiki/File:Europa-moon-with-margins.jpg https://en.wikipedia.org/wiki/File:Io_highest_resolution_true_color.jpg http://photojournal.jpl.nasa.gov/catalog/PIA00716 http://photojournal.jpl.nasa.gov/catalog/PIA02308
Font ("Alien League")	https://www.fontspace.com/iconian-fonts/alien-league
Font ("Unispace")	https://www.1001freefonts.com/unispace.font
Water disp./normal map	https://www.youtube.com/watch?v=qgDPSnZPGMA
Music	https://www.dl-sounds.com/royalty-free/category/atmospheric-meditation/space/

Sound effects <http://soundbible.com/>
 <http://www.orangefreesounds.com/>

Information/Tutorials:

Learn OpenGL <https://learnopengl.com/>
OpenGL SuperBible <http://www.openglsuperbible.com/>
Real-Time Rendering, 4th edition <http://www.realtimerendering.com/>
"Game Physics" articles <https://gafferongames.com/categories/game-physics/>
Quaternion tutorial <http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/>

Shockwave loosely based on http://empire-defense.crystallin.fr/blog/2d_shock_wave_texture_with_shader
 (no longer exists; accessible via archive.org)

Procedural lava shader based on <https://www.shadertoy.com/view/lsIXRS>
Lens flare tutorials <http://www.emagix.net/academic/item/lens-flares>
 <https://www.gamedev.net/articles/programming/graphics/lens-flare-tutorial-r813/>
 <https://www.opengl.org/archives/resources/features/KilgardTechniques/LensFlare/>

Particles/Transform feedback <http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html>
 <http://www.mbsoftworks.sk/tutorials/opengl3/23-particle-system/>
 <https://open.gl/feedback>

Complete keyboard / mouse control reference chart

Bindings that are only used during development and for debugging are marked **red**.

Quickly hit the ^ key (caret key, above TAB) three times in a row to enable the developer keys. If successful, an on-screen message is displayed.

Move mouse	Rotate camera in cursor-capture-mode (default) If cursor-capture is off, use mouse-buttons in addition
W,S,A,D,Q,E	Rotate player around its local axes (actuate control thrusters)
Space	Engage main thruster
LMB, C, PgUp	Auxiliary system: Prograde (orient player towards flight direction) Hold Shift to perform turning without aligning up-direction (faster)
RMB, Y, PgDown	Auxiliary system: Retrograde (orient player against flight direction) Hold Shift to perform turning without aligning up-direction (faster)
MMB, X, Home	Auxiliary system: Normal (orient player upright)
B	Drop EMP bomb
O	Toggle compass
H	Toggle HUD
Shift-R	Reset level
R	Reset to last checkpoint
P, Pause	Pause game
U	Toggle cursor capturing (brings mouse cursor back)
Esc	Menu
Alt-Return	Toggle full screen mode
F3	Toggle FPS-display
F5	Sound: Shift-F5 / F5: set volume; Ctrl-F5: mute on/off
F10	Brightness: Shift-F10 / F10: decrease / increase; Ctrl-F10: reset
^ (caret key)	Hit 3 times within a second to enable developer keys
Ctrl-N	Toggle normal mapping on player's ship
Up,Down,Left,Right	Debug camera control (rotate camera in fixed steps)
F	Freeze player (for debugging)
K Shift-K Ctrl-Shift-K	kd-tree culling report (console) dump kd-tree to console rebuild kd-tree
Ctrl-H	Overlay PhysX terrain heightfield (debug compilation only)

Ctrl-Shift-H	Cheat: gain additional helium
Shift-B Ctrl-B	Cheat: gain an additional EMP-bomb Cheat: launch EMP-bomb disregarding groundspeed
G	Toggle gravity on/off
V	Toggle player angular velocity dampening on/off
Ctrl-T	Toggle terrain adaptive tessellation
T	Toggle camera tracking player
L	Toggle camera free-look (if cursor-capture is off)
Ctrl-L	Toggle LOD switching
1,2,3,...,9,0	Toggle rendering of selected elements Ctrl: toggle rendering of different model types
Ctrl-I	Toggle invincibility cheat (no player collision)
Shift-I	Dump timing report to console (debug compilation only)
I Ctrl-Shift-I	Dump debug info to console Detailed per-object debug info
Shift-M	Toggle multisampling
Ctrl-S	Save checkpoint (also if not landed)
Numpad Enter Ctrl-Numpad Add	Toggle display of hitboxes Shift: render culling boundaries of objects Ctrl: visualize kd-tree Select kd-tree cell to visualize
Numpad keys	Reposition player (Shift,Ctrl,Alt control granularity)
Shift-Numpad Sub	Initiate player separation, even if not landed
Ctrl-Alt-Numpad	Move window
F1	Toggle wireframe mode
F2	Toggle backface culling
F4	Toggle view frustum culling
Shift-F4	Toggle additional per-object view frustum culling