Proposal for a 3D Racing Game: Froschidrive 2

Inspired by the lecture *Computer graphics* I already developed a WebGL based 3D racing game called *Froschidrive*[1]. With the internship I see an opportunity to be able to continue working on my project in order to realize correct 3D intersection tests, more complex shaders, as well as diverse game modes and storage efficiency. I would prefer to work alone to get a more broad insight into game development and because I want to keep the existing workflow which would include the development on my own server in Javascript.

Existing Workflow

For the modelling, model hierarchies, texturing and custom properties I use Blender 2.8 and some basic 2D image manipulation programs like paint.net or Gimp. From Blender I export the compressed data as packed .gltf[2] and try to read it with a self-written importer (which has to be extended) and integrate it in my .js WebGL pipeline. The pipeline only includes glMatrix[3] for high performance linear algebra functions as dependency. For debugging I mainly use the native Google Chrome Debugger.

The entire project can be found under the following URL: `https://git.scc.kit.edu/udfiy/froschidrive_2`.
The ~~currently developed~~ released version can be found here: `https://t4ce.de/games/froschidrive2/`.

End Goal

Implement a 3D-Racing game which is capable of full 3D movements, collisions, the Phong reflection model and supports additional effects such as transparency, reflective objects, shadows and emitting particles. It runs on any current chromium based browser online, is storage optimized for a fast download and unlike other engines, it has as little unnecessary overhead as possible. It also provides different game-modes like a local multiplayer for more fun experience compared to the previous version.

Milestones and personal Report

1. **Define requirements, refactor code and implement prerequisites**
   (a) Submit second proposal, adjust after consultation
      i. The second proposal was more sound. Added tasks are shadows, point light sources, particles, morphs and a basic level of detail method. The Texmaker was used to create the document. The version control over the KIT SCC git only worked since last week because the following sentence confused me: *To log in, please use your AD account in the form xy1234.* I could not find an AD account anywhere and besides, my access data was not in the form xy1234. It is now running, but I have not yet been able to add a supervisor to the project because I do not know an official git name.

   (b) Show the loading process, Enable fullscreen-mode, FPS
      i. It turned out to be tricky to display a div container over an HTML5 canvas element. Advice on using the z-index didn't help, the solution was *position: absolute* for the div and the canvas too, and *box-sizing: border-box* for only the canvas. Scroll bars were also annoying but removable.
      ii. The full-screen mode has to be initiated by direct user input and it was difficult to let the cursor disappear simultaneously, especially for Firefox. In the meantime, however, it runs stably in all browsers and a separate .js file for GUI operations was created.
      iii. For the FPS it was important to use requestAnimationFrame to allow a dynamic frequency of the monitor. It turned out that the carrot only rotates about half as fast with a 60Hz screen as with 144Hz, which had to lead to a code restructuring. While now all graphic elements are calculated frequency-dependant, the physics are controlled over a fixed clocked interval, which should not change on different devices. For this reason, an additional calc FPS is now displayed.

[1]`https://www.t4ce.de/games/froschidrive/`
[2]`https://github.com/KhronosGroup/glTF/blob/master/README.md`
[3]`http://glmatrix.net/`

(c) Code refactoring from scratch, increase performance

   i. It was very time-consuming to remove everything game-specific from the froschidrive 1 code, such as hard-coded rolling rock animations, 20 dimensional arrays and a huge number of state variables and security measures. Instead, I've created classes for game objects which contain the state variables and use dictionarys and teir keys instead of arrays and huge switch statements.

   ii. The performance could already be optimized when splitting into physics and render loops, especially because object lists are now updated less frequently but more evenly on different devices. However I assume that the weak points will show up later.

(d) Extend the .gltf-importer (childs, lights, material properties)

   i. The new Blender 2.8 version now supports gltf 2.0 natively, which is why the exported code brought some changes: The frame buffers were further divided, some attributes were renamed, child hierarchies expanded, and the standard material properties changed, as did the export settings and Blender itself.

   ii. My new gltf importer can now read in the object color, recognize metallic and roughness and merge vertices from different child nodes before the model is passed on. Object clones are no longer defined manually using extra attributes, but a clone looks for its parents if it does not have its own vertices. A clone can also be created from merged objects and can be used by other parents and their children.

   iii. This procedure has drastically reduced the storage requirements for round objects such as gearwheels (approx. 30%), but leads to a significantly increased modeling effort and seduces to use too many vertices. By using textures with a low bit depth, the storage effort could be halved. The current version including some low-poly objects is just 1.3 MB in size, which can be downloaded in Germany with an average bandwidth of 24 *Mbit/s* (3 *MB/s*) in under half a second (122ms in Taiwan).

   iv. The sun direction and point light sources are also recognized and later converted to game objects and passed to the fragment shader. The new blender sun direction is already in use.

(e) Phong reflection model, influence it by .gltf material properties

   i. The Froschidrive 1 shader, which comes from a YouTube tutorial by Indigo Code, has been expanded to include the specular term. First I used the lecture's reflection formula, but later replaced it with GLSL reflect and checked whether the lambertian part is positive like in the shaders of Uni Marburg[4].

   ii. Once the shader was set up, influence its uniform variables by the importet material properties was not a big deal. On the other hand the variables itself were problematic because I found no clear definition how to use metallic and roughness factors for the phong reflection model. At the moment I use: $(1-roughness)*(1-metallic)$ for the specular factor and $(1-roughness)*100$ for the exponent, what definitely has to be fine tuned if different materials exists.

(f) Design new cart and .gltf test-environment in new Blender 2.8

   i. For the new kart, I used a more realistic frame from a real kart and introduced axles for the front tires that move with the steering. All tires are now clones from the front right wheel, which itself consists of only one tire segment, which is rotated 32 times in a circle. I modeled a tire tread for fun, which is why a tire currently consists of 4352 triangles, which will be replaced by a texture in the future, but I'm happy that everything still runs smoothly :)

   ii. To enable later levels of detail, froschi now has a low-poly brother that has no morph properties and required a revised version of the UV mapping. This is exactly how low poly eyes were made, which should be able to move independently of Froschi. The plan is later to display the other players permanently in the low-poly version in multiplayer mode, since Froschi will probably have the most vertices. The distance could be used to decide which level of detail could be used for other objects.

   iii. The map now has a primitive sand texture with less than 5kb memory size. However, it is very difficult to create a small texture without recognizing repetitions as a person. Perhaps noise or procedural textures would be interesting as future projects.

   iv. The map now consists of ramps, some test objects with transparent and metallic materials and a house in which a point light source is located. With the new map, all innovations should be able to be tested for the next two milestones.

[4] `https://www.mathematik.uni-marburg.de/~thormae/lectures/graphics1/code/WebGLShaderLightMat/ShaderLightMat.html`

2. **Improve Froschidrive in terms of graphics**

   (a) Create a single fixed skybox, reflect skybox in fragments

      i. A Cubemap seemed to be ideal for the Sky, which was explained in various tutorials[5]. The biggest challenge was to prevent it from moving with the camera and unexpected behavior of glMatrix functions such as different angles in degrees and radians and a less well-documented lookAt function. I also learned a lot about quaternions and mostly manage to put the transformations in the right order. The Cubemap itself consists of six individual images that are generated using an HDRI-to-Cubemap tool. At the moment the HDRI comes from the Internet, but I found out how a panorama image can be rendered with Blender, which is the intention for the future if all the 3D-assets are finished.

      ii. For the reflection of the Cubemap in the fragments, I reflected the camera direction on the fragment's normal to get the cubemap texture coordinates. I really liked the results, but since I don't want to automatically render a Cubemap for each reflective object, reflective elements will be small and rough.

      iii. In the meantime I noticed that it makes more sense to move some calculations from the vertex shader into the program, e.g. projection, camera and world matrices.

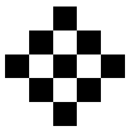
   (b) Use a directional light source to create shadows

      i. For this task I used a Frame Buffer Object as well as an ortographic camera for the first time. At the beginning it was not entirely clear to me that you had to reset the FBO to null after binding and that you can clear color and depth buffer independently. In addition I had to adapt some values as the bias to get the shadow finally to work for my scene.

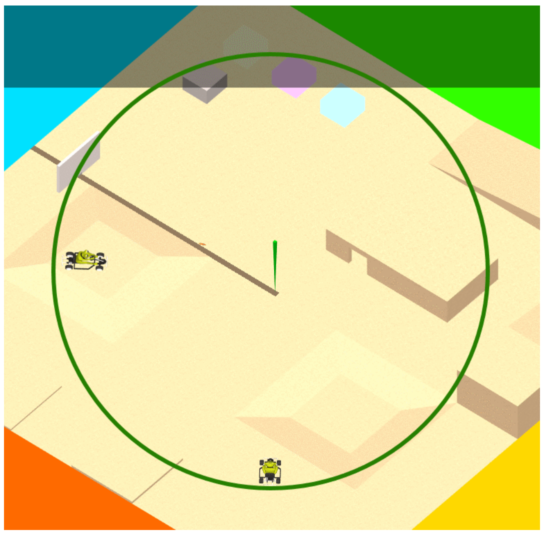
      ii. Although I was initially happy with **shadow acne** since the projection finally worked, I tried for a long time to find the optimal bias. After trying to set this per object, I found that there are also differences within an object and that a high bias leads to so called **Peter Panning**. After a quick search I found the following formular: $biasFactorUniform * tan(acos(dot(n, l)))$.

      This solved the problem, especally for round objects, but not on my smartphone. After a much longer search, I found out that the float **percision** was the problem, because it can differ on various systems. I have therefore set the percision to highp for the bias and depth calculations part of the code and set it to lowp for the remaining fragment shader code.

      iii. After the bias is set automatically, I have tried to **smooth** the resulting edges from the depth buffer. The main idea was to look for a certain amount of neighbour-depth-pixels and to take them into account. After trying different patterns, I decided to include 8 neighbors. 4 of them in the corners with a distance of 1 and 4 more directly above and next to the pixel with a distance of 2 as you can see on the image →



      iv. Higher **resolutions** always tend to way better results. However, this is only effective if the area watched by the orthographic light camera is small. The first approach was to move the light camera with the player camera. This led to two problems: firstly, it didn't really make sense to calculate the area behind the camera and secondly there were flickering effects. The latter could be remedied by incremental camera movements, which are only carried out when a certain distance is reached. For the first problem I decided to move the light camera on a circular path according to the rotation of the player. This was not so easy, especially since I wanted the rotation of the light source to be dynamically adjustable in Blender. In the picture on the right you can see one valid light camera position for two different cart positions with different yaw-angles.



      v. After that I was not that happy about shadows suddenly disappearing in the distance. So I decided to introduce an additional **XL-Shadow** which is indended to prerender the whole scene once for all

[5]https://webglfundamentals.org/webgl/lessons/webgl-skybox.html

players with a lower resulution compared to its size. At this point, object orientation really paid off and it was not difficult to create another shadow with different properties. As a bonus extra, I let the nearby shadow slowly and **smoothly** change into the XL shadow. For this transition I mainly used the distance between the player and the fragment to decide how much influence each shadow has.

vi. During the whole process I was disturbed by the fact that it seems necessary in WEBGL to always combine a frame buffer object including a depth texture, with an additional unused color texture. So I decided to use the color texture for transparent shadows in the next step.

(c) Use blending to show multiple semi transparent objects

i. Activating the blending was easy and looked good, but only from one side. Objects that were behind transparent objects partially disappeared, which is why I now sort the objects according to their distance from the camera. For the next milestone, it is optionally planned to merge nearby objects into one and sort the individual triangles.

ii. Then I created a palm tree with leaves, which have a transparent background and should also cast an appropriate shadow. It wasn't easy to create a palm tree bark texture. Since I couldn't find a suitable texture for the leaves online, I modeled the leaves with a gradient and ended up rendering the result as a texture. This also had the advantage that I was able to significantly reduce the number of semi-transparent pixels on the edge of the leaves. Since the sorting does not change from the point of view of the light source, all objects are sorted by default from the light and adapt each frame to the individual player. A big optimization process started: At first I was unsure whether I should simply discard transparent fragments and use the depth buffer, or better use blending and transparent colored shadows. I chose the second option because the edges looked much smoother and it was impossible to find a suitable limit to discard, especially since the textures always had semi-transparent pixels. With the previous blending function, the leaves looked thin and had a strange white edge shadow, which is why I disabled culling and adjusted the blending function for the leaves as follows:

$gl.blendFunc(gl.SRC\_ALPHA, gl.ONE\_MINUS\_SRC\_ALPHA);$ // Semi-transparent materials
$gl.blendFunc(gl.ONE, gl.ONE\_MINUS\_SRC\_ALPHA);$ // Leaves & prospective plants

At this point I was very happy with the color texture, because I can specify a minimum level of detail to automatically smooth the shadows. To set a minimal mipmap level of detail and for other convenience I upgraded WebGL to version 2.0. I can't smooth the depth texture so easily, however the depth texture is more precise for hard edges. For the combination of depth texture and color texture, I finally decided to choose the minimum brightness per color channel instead of always giving priority to the depth texture to avoid borders around shadows. In addition, the opacity of a transparent object affects how dark the shadow becomes, but it never gets darker than that of a non-transparent object.

(d) Create point lights without shadows for later effects

i. I decided to use the following attenuation function for the point light sources:

$$\frac{L_c}{a_0 + a_1 * d + a_2 * d^2}$$ with Light color L_c, Fragment to light distance d and atenuation factors $a_{0..2}$

Unfortunately, I had to find out that the attenuation factors are not that intuitive to use, but I found tables with suitable values online. In order to give the impression of a shadow and to prevent walls from being illuminated on both sides, I decided on the following self-explanatory criterion:
$if(fragPlusNormalToLightDistance < fragToLightDistance)\{calculate\_attenuation();\}$

(e) Unexpected fates

i. On May 19, I was no longer able to access my server via sFTP due to Telekom problems. The attempt to start a server locally failed due to security policies of loaded textures from google chrome. As long as there are only failures of one to two days, the effects are negligible, but in the future it would make sense to think about a local development environment.

ii. From May 27th to May 28th, google chrome updated itself. I was therefore very surprised when I suddenly received warnings that I had never heard of before. But I was even more surprised when I restored the last version via Git and continued to get warnings. In the end it turned out that I had not set the active textures correctly in the render loop, but it was difficult to locate the bug because it could not be traced with Git and there was no information about the line location.

3. Drive over mountains and valleys

(a) Remember and implement barycentric coordinates for triangle intersections

i. For correct 3D movements it is necessary to read out information about the triangles of the ground under the kart. I decided to use projected (y-up) barycentric coordinates to first check if the player position is within a triangle and later to get the intersection position, especially the y-coordinate. For this purpose I implement a triangle class, which provides the center of a triangle, the area and methods for calculating intersection points.

ii. These so-called collision triangles of an object are only calculated if the object in Blender has an extra attribute with the name *sphere*. The sphere attributes were used in the first Froschidrive version to determine collisions between round objects. Now they are reused to determine how close the player is to an object. If it is close enough, the individual collision triangles are checked for collisions.

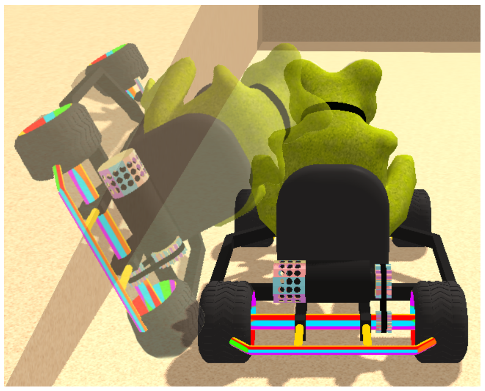
iii. I decided to split objects with sphere-attributes into ground objects with friction factors and normal objects. The latter are only checked for a maximum of one intersection and collide immediately when they come into contact with the player. On the other hand, it is possible to drive on ground objects, with several positions being checked at the location of the tires. But it is also possible to collide with a ground object if the height difference between the kart and the triangle becomes too large.

(b) Use the hitpoint for new car position and rotate according to its normal

i. The approach of using the normal of the ground triangle for the rotation of the kart was ideal for smooth surfaces. With artificial objects such as on a ramp with a certain slope and hard edges, there were always tires that sink into the ground. For a long time I tried to find a good transition factor depending on the speed, but to no avail. I therefore decided to determine a valid position for each tire.

(c) Detect collision, free fall and unwanted states, use multiple hit-tests

i. The issue with valid tire positions turned out to be very difficult. Especially when individual tires are hanging in the air, which is why I now interpolate between the tire positions to determine a pseudo center of gravity. Unfortunately, there are still many problems if there are two or more tires in the air. For example, if the two right tires are hanging in the air, the kart should turn over the edge and fall over, which would be an unwanted position. I looked at *Mario Kart 64* and found that in this case the kart is simply moved to the right to avoid this unpleasant case. I also found that side abysses are often delimited by bands and edges at the ends of jumps were given acceleration fields to avoid slow tipping. I think I will include some of these ideas in the later game design.



ii. For better and timely collision detection, I also check other positions in the direction of the car movement. The behavior in the event of a collision is essentially to travel in the opposite direction and to reduce the speed. Another state feature is that the kart can roll down hills depending on its rotation.

iii. In order to realize the free fall I gave each kart a boolean freeFall attribute and a fallingTime which is organized via the javascript internal date functions. I used the following formula for the calculation: $h = \frac{g * t^2}{2} = \frac{9.81 * fallingTime * fallingTime}{2}$ with location factor g, distance h and time t. The free fall is triggered when all tires are in the air and stops as soon as at least one tire is on the ground.

(d) Collide with different objects including other players, optimize it

i. This task required some organizational effort in relation to the source code. It was now really necessary to create several carts with their own attributes, each of which should have an individual control. The close shadow had to be calculated depending on each player, as was the final picture. To see if everything worked, it made sense to implement the split-screen mode from the next milestone. Fortunately, I had already thought a lot about multiple players in the past and apart from some problems with active textures and deleting buffers, it is now possible to play with up to four players at the same time.

ii. Unfortunately, I noticed a limitation during this time: It depends on the keyboard how many keystrokes can be registered at the same time. Sometimes there are only three, which is very problematic because each player can already use four buttons. It is therefore possible to block other players by pressing all keys at the same time. Should it become problematic in normal game operation, one could think about accelerating automatically.

iii. Each kart has a hit polygon for collision with other players. If the hit polygon of your own kart is hit you get the collision direction of the other player. Depending on the angle between your own movement and collision direction, the speed of both karts is influenced.

(e) Affect friction factors depending on the surface

i. Since it is only possible to move over ground objects, I had to support several ground objects for this task. The difficulty literally existed in edge cases. For example, the kart should collide if a tire is outside the current ground. But what if the tire is on another ground or multiple grounds at the same time? So I had to sum up all tires touching a ground and if that number is smaller than the number of checked tire positions the cart will collide. Each kart now also has a coefficient of friction, which increases per tire that is located on a special ground. This enables a more realistic transition between different surfaces. At the moment there is a stone ground with the least friction, sand with a medium and water with the highest friction.

(f) Deform parts of the cart after collision via morphs

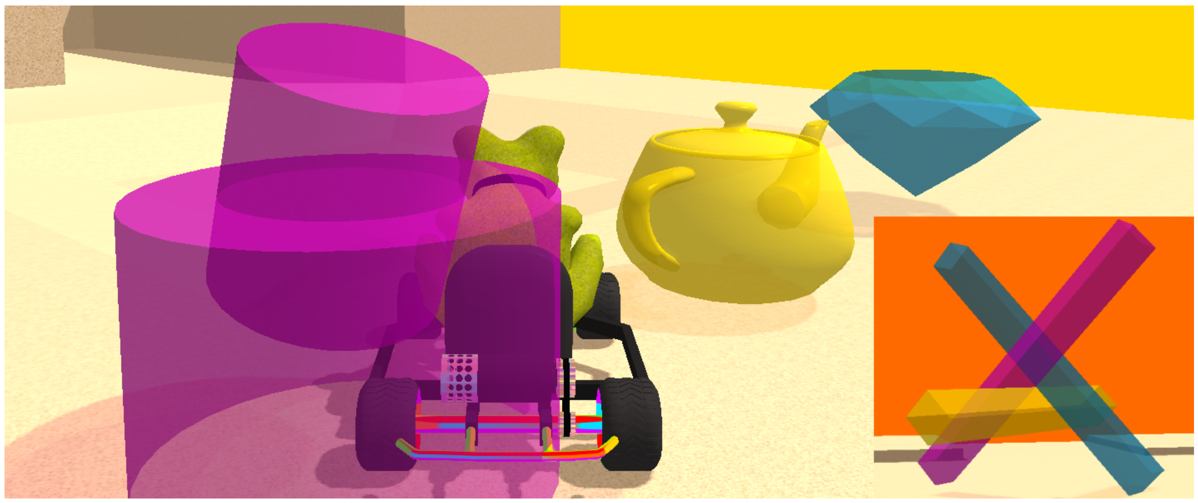
i. Like Froschi, the bumpers now also have morph targets. Depending on where the kart collides, either the left or right side of each bumper deforms. Since the back bumper is only a rotated clone of the front bumper, left and right morph had to be swapped.

(g) **Extra**: Depth Peeling for order-independent transparency

i. As mentioned in the review of the second milestone, I wanted to look at blending techniques for transparent objects that do not require manual sorting. After a short research I discovered the so-called depth peeling, for which there is a very good white paper from Nvidia. The basic idea is to rasterize the scene for transparent objects several times and to use the depth information of the last layer to pay attention to the texel behind it in the current layer.

ii. To do this, I first had to render the opaque objects to a texture. Then I created an additional shader for peeling layers, each of which contains the deep buffer of the opaque and previous peeling layer. I also created various pre-multiplied blending modes. For each layer I used a separate frame buffer object, which textures resoluions are updated every second when the window size is changed. As a result, each frame buffer object contains a color texture of the corresponding layer.

iii. In a second step, I wrote a shader that receives the opaque and all transparent peeling layers and does the blending. The resulting image is displayed on a full-screen rectangle consisting of two triangles.


Screenshot of various transparent objects that are problematic with conventional object-based sorting

4. Design a tropical map and diverse game modes

(a) Create animated particles to pretend wheel traces in the sand, dust behind cart, water splashes when entering water and fire

i. Once again, a lot of organizational effort was required for this task. The first thing I wrote was a particle class, which should take care of the physical properties such as ejection speed or weight. In order to have the particles emitted at different locations, it was necessary to introduce particle systems that each manage a large number of particles.

ii. I quickly realized that the previous *gl.drawElements* call was unsuitable for particles and I dealt with instanced rendering. In order to be able to transmit different positions or world matrices to the particles, it was necessary to define new attribute buffers and to use the gl.vertexAttribDivisor. Since each particle should change its appearance via a texture atlas, it was necessary to transmit a uv offset to the vertex shader in addition to the world matrix. Since there are textures that are just supposed to fade out, I also added an opacity value per particle instead of saving the same texture with different opacities to the texture-atlas what again reduced the data overhead.

iii. I found it useful to summarize particle systems of the same type before they are rasterized to further reduce the amount of draw calls. This is now done by an almighty particle system manager.

iv. After that I only had to paint some texture atlases. But it turned out to be one of the most time-consuming tasks ever! The first plan was to render single particle images with blender and to combine them into a single texture atlas (sprite). I was horrified to find that there was no reasonable program online that could do this with a variable number of images in with. So I decided to implement it: `https://t4ce.de/tools/merge_images/`. Feel free to use it for your own Projects :)

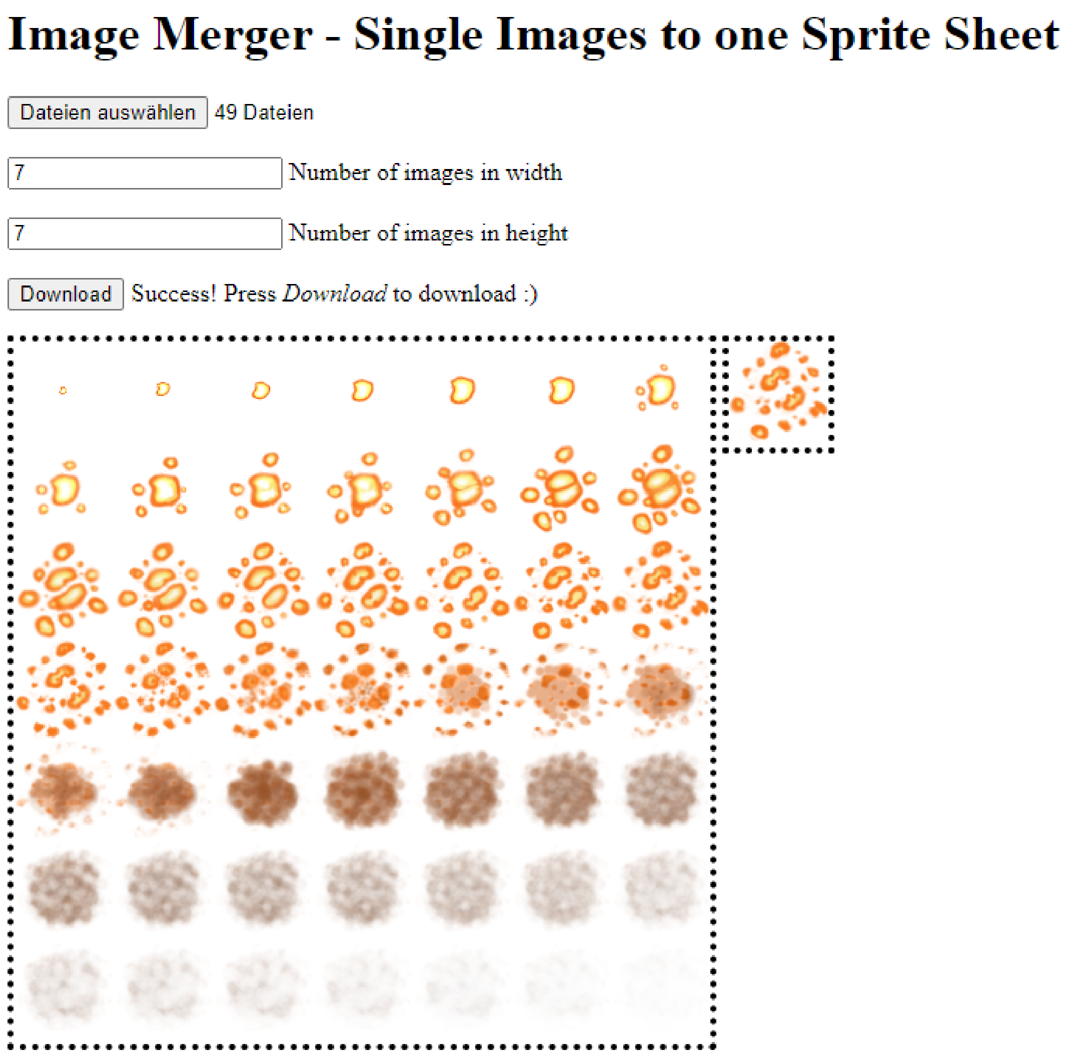


Image Merger - Single Images to one Sprite Sheet

Dateien auswählen 49 Dateien

7 Number of images in width

7 Number of images in height

Download Success! Press *Download* to download :)

v. After a few unsuccessful attempts with the new blender 2.8 particel systems I finally decided to draw every single image with Paint.NET. As you can see above the game now includes fire particles for the vulcan and in addition dust particles behind the cart, water splashes at sea level and dark wheel traces behind the frontmost wheels, which will be washed away if they are under water.

(b) Design a tropical map, that represents the developed techniques

i. I started with a huge sand island and immediately had performance problems caused by the triangle intersection tests. The first idea was to split the island into several pieces, which caused shading problems at the transition points and increased the number of objects and draw calls at the same time. Instead I decided to only check the nearest triangles with less than a certain distance to the car what worked pretty well. After watching my roommate drive across the mountain to cheat, I decided to exclude the volcano in the middle of the map from the cutting tests and to collide instead.

ii. Collision was generally a big topic because I had nearly no overlapping grounds in my test environment. But it absolutely made sense if you are under water or drive on a rock which is lying on sand. Because it is also interesting for the particles to know, every tire now has a list of grounds on which it is located.

iii. I decided to take over the sea from Koopa Troopa Beach from Mario Kart 64 and move a large surface up and down as sea level to create the illusion the water would run onto the beach. However, the resolution of Mario Kart 64 is much worse than in Froschidrive, which is why you saw the horizon moving up and down. So I tried to divide the surface of the water into movable and immovable, distant areas, what created an unwanted gap if the water level changes. So I just created a morph for the vertices near the coast, which solved the problem with the horizon. On the other hand, it is now much more difficult to determine how high the water level actually is. With some arithmetic and a sine function I am now reasonably satisfied.

iv. In thinking of the environment and to show off my depth peeling, I decided to introduce an additional game object: a glass bottle from the *Hüpfer Bräu* brand. When you touch it, a unique morph activates, which destroys the bottle. Your tires will lose air, the carrot bonus will decrease, as will the speed. After the bottle is destroyed, the radius in which the player experiences these negative effects increases.

v. As in the first Froschidrive version, I wanted to have movable rocks. They have also a movable hitbox und will rotate automatically and calculate their position dynamically instead of being manually described. They are intended to be an obstacle to the shortcut over the volcano, but can also be used voluntarily to annoy other players by kicking it down the hill.

(c) Implement basic discrete levels of detail for distant objects

i. It took me a long time to realize that in my case it is more worthwhile to merge objects in order to reduce the draw calls instead of using multiple levels of detail. Nevertheless, there is now a low poly Froschi version with low poly eyes that are displayed from a certain distance since Froshi has the most vertices and two morphing targets. In addition, bottles and carrots are only shown from a certain proximity. For the future I plan to focus more on instaced rendering.

(d) Implement a local multiplayer optimized for 2 players

i. I already completed this task in the last milestone :) However I had to *Math.floor()* the player's height to prevent scrollbars from schowing up.

(e) Stop the time and prevent in-game cheating

i. Since I put a lot of effort into separating the display from the playing time, I hoped that the Javascript intervals would now work better. It turned out that Firefox's setInterval runs slower than in chromium-based browsers. Therefore, the vehicle position updates less frequently and the kart moves less, which is a clear disadvantage. I think it makes more sense to point it out in the main menu than to exclude firefox from the game or to write my own time function.

ii. In order to fight ingame cheating, I placed 5 checkpoints on the map this time. It is not possible not to touch them. If you touch them in the right order your player status will increase. If the status is high enough and you pass the finish line the lap counter will be increased. At the same time, the player position and rotation are saved when touched. If a player gets stuck under the map, he will be reset to the last valid position. This case now only occurs when crashing on steep slopes.

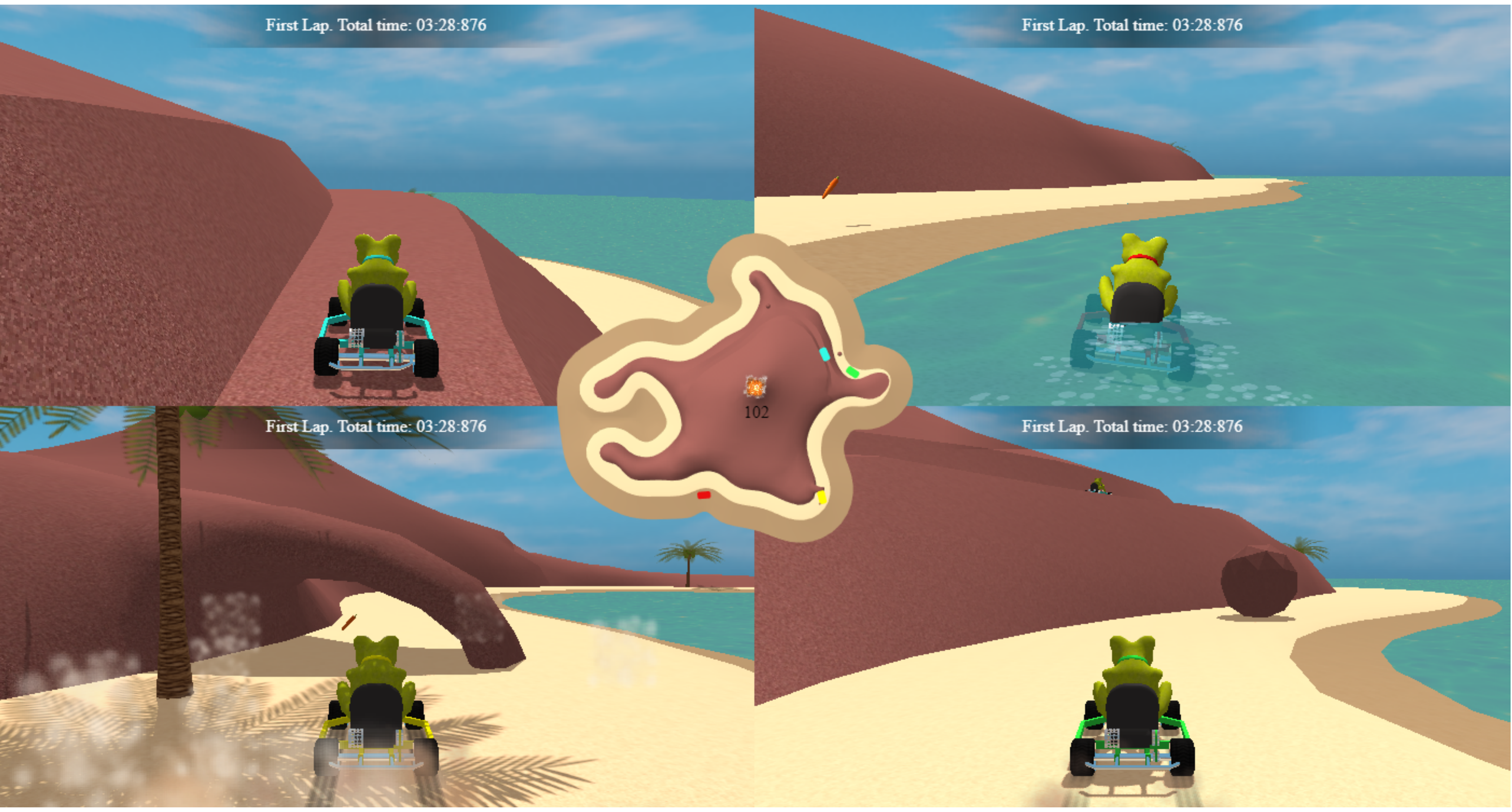
(f) **Extras**

i. **MiniMap**: To give the players a little more orientation, especially in multiplayer mode, I added a mini map. Depending on the number of players, it is placed in the center or in the corner of the screen and shows opaque objects, vulcan fire particles and enlarged black hit boxes of the players.

ii. **Selectable player colors**: To give players even more guidance, each player can now choose their favorite color. I used an html color input element for this purpose. Depending on the browser, a color wheel is even displayed, which enables a compatible and quick selection. The player color will affect all objects to which I have assigned a customizable material in blender: Froschi's ribbon, the frame of the kart and the timing belt. In addition, the kart is shown in the corresponding color on the minimap and players are automatically assigned colors when they are added.

iii. **Menus**: The game starts with a main menu where players can rename themselves and choose their colors. A plus button can be used to add new players and remove them again with minus. It is also said to offer a little help. You can switch from the main menu to the game or to the high scores. During a game it is possible to pause the entire game with the pause key (on the keyboard). A pause menu will then appear, from which you can restart the game or return to the main menu. Throughout the process, the players keep their names and colors. In addition, the .gltf file only needs to be loaded once, which significantly improves the flow of the game.

iv. **Highscores**: As in the first version, there is also a global highscore table this time. It contains one column more because you now have to drive three rounds instead of two and is protected against trivial hacking. The highscore table can be accessed from the main menu and adapts dynamically to the screen and the entries it contains. A non-proportional font is used to try to keep the text in the same length. In split-screen mode, the data is transmitted during the game and the rank is already shown to the players, before everyone has reached the goal. In split-screen mode, every player is registered in the database. Looking forward to seeing your names there again :)


First Lap. Total time: 03:28:876
First Lap. Total time: 03:28:876
First Lap. Total time: 03:28:876
First Lap. Total time: 03:28:876
102

Screenshot of the final Game. Play it at `https://www.t4ce.de/games/froschidrive2`

Tools

- Chrome, *new* Edge, Firefox
- Blender 2.8, paint.net
- Notepad++, LaTeX
- HDRI to Cubemap by Matheowis

Libraries

- glMatrix
- WebGL

Reference Games

- Froschidrive 1
- Super Mario 64
- Mario Kart 64
- Mario Kart 8

Tutorials

- `https://webglfundamentals.org/`
- `https://www.khronos.org/gltf/`
- WebGL Youtube playlist by Indigo Code
- OpenGL Youtube playlist by ThinMatrix
- Shader from Uni Marburg[4]
- Shadow Improvements
- Common Mistakes explained by Khronos

Papers, Journals and Books

- Marrin, Chris. "Webgl specification." Khronos WebGL Working Group 3 (2011).
- Robinet, Fabrice, and P. Cozzi. "Gltf—The Runtime Asset Format for WebGL, OpenGL ES, and OpenGL."
- Kellogg, W., Jason Ellis, and J. Thomas. "Towards supple enterprises: Learning from N64's Super Mario 64, Wii Bowling, and a Corporate Second Life."
- Everitt, Cass. "Interactive order-independent transparency." White paper, nVIDIA 2.6 (2001): 7.