# Developing a Jump'n'fly Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Julius Elias Oliver Major Adrian Niewiadomski Kai Schmitz


## 1 The Game

*Naspar* is a jump'n'fly game that was developed to train some basics of game programing and computer graphics. In the game you steer a spaceship with your mouse over an obstacle course to reach the goal as fast as possible while collecting powerups and score bonuses.

## 2 The Engine

### 2.1 Architecture

The game is programmed based on a game engine that was developed by our team.
The base class to all objects that can be seen in the world is the `DrawableObject` class. It contains geometries, textures and material parameters as well as methods to load and assemble the parameters from disk and draw the object. Derived from this class are the so called *data-classes*, which hold additional but static data to the drawable objects. For each data-class there exists an associated *dynamic class*, which has a pointer to its data-class and all dynamic parameters. There are two main reasons for splitting up the static and dynamic data of an object in two classes: A better memory consumption and the more important aspect, that it allows for a fast copying of objects (cf. Section 2.2).
The core of the engine's architecture is the `Game` class, which implements the `Module` interface for easy integration in the lightweight framework, that our `main()` function delivers. The `Game` class holds all data that are related to the Game, such as the player's `Car` object, the `Course` object, the score and timer and some more. It also detects pressed buttons and knows how to calculate the game physics, for example the self made collision detection mechanism, which tests if the car's and an object's two-dimensional rectangular hitboxes overlap. The class also implements the `Module`'s `draw()` function, which simply delegates the task to an object of the `Renderer` class, whose function is to simply render the game.

External libraries that are used in the implementation of the engine are *GLFW* (window creation, input polling, multithreading), *pthreads* (multithreading), *TinyXML* (loading settings, object parameters and track data from disk) and *ACGL* (drawing to OpenGL, graphics management).

### 2.2 Multithreading

The game's source code offers a mechanism that allows the user to switch between several *multithreading modes* while compilation. To do so, you can edit the `MULTITHREADED` macro in the Mutlithreaded.hh file to be one of the following:

- `THREADS_NONE` will setup the compiler to create a singlethreaded executable. Game physics and graphics will be called from the only thread.
- `THREADS_POSIX` will setup the compiler to create a multithreaded executable. The main thread will handle the graphics, while another thread will be created that handles the game physics. For thread creation and synchronization the pthreads library is used.
- `THREADS_GLFW` does the same as `THREADS_POSIX`, but uses GLFW for thread creation and synchronization.

The synchronization of the threads takes place within the `Game` class. It uses two mutexes, the so called `structMutex` which locks the class object when it is changed, and the so called `waitMutex` which locks the locking of the `structMutex`, so it serves as a barrier that prevents a thread from locking the structure again after unlocking it if the other thread is already waiting.
To increase the execution speed and prevent the graphics thread to hold the mutex lock longer than neccessary, the `draw()` function makes a temporary copy of the `Game` object and then draws it after releasing the original's mutexes. This mechanism requires a fast copying of `Game` objects, which is obtained by splitting dynamic from static data as depicted in section 2.1.
The result of the aforementioned implementation results in both better CPU usage and a higher frame rate.

### 2.3 Problems

While programming the game we encountered some problems, of which we want to explain a few crucial and extraordinary examples.

- When using the singlethreaded mode at low frame rates the collision detection does not work properly, because the car might simply jump over track objects at high speed without having their hitboxes overlap at any time. The problem does not appear while multithreaded execution, but it does however persist in singlethreaded mode on slow machines.
- As we included multithreading capabilities to our program, we encountered some objects that moved in wrong directions or even not at all. We found the problem to be a consequence

of the better physical resolution, that made little `double` values for speed and delta time round to 0 when casted to `float`. The solution to the problem was to limit the physical resolution by inserting short delays in the physics thread. We found a value of at most 10000 physical frames per second suitable, so we inserted a delay of 0.1 ms in every turn of the physics thread.

- Another hard to find bug were random program crashes at the termination of the GLFW context. As we found out, the GLFW is not thread safe, which means that calling GLFW functions from more than one thread in the same context creates undefined behavior.
  The problem could be solved, by deleting two calls to `glfwSetMousePos()` from the physics functions, but to provide the same functionality as before, we needed to reimplement a big part of the input polling.

- Finally, we encountered a problem that the graphical frame rate of the game bursted down with the implementation of some of the last features. Debugging the code revelaed the problem to be a result of doing things for the first time and therefore implementing a suboptimal program design. With our architecture the `Renderer` calls draw functions for over 1000 objects with a huge method call overhead of 3 to 5 method calls and 6 matrix calculations per object.
  The problem could not be solved in time, so it still remains in the final release of the game, because solving it would mean to start over again and implement the game from scratch in the most parts. However, we have been working on a possible solution to this issue, a revised program design. In the better design the `DrawableObject` class does not only save the static data of an object, but also its model matrix. It provides methods to move, rotate and scale the object, which then affect the model matrix directly. The other classes are built on top of that. The `Renderer` class keeps track of all objects in a list of pointers to them. To draw a game scene, the `Renderer` would simply call the `draw()` method for all elements in its list. Therefore the result would be one method call per object, less passed parameters and only 4 matrices to be calculated within the method.
  We can not tell how much speedup we would gain with the new design, but we expect frame rates of at least 60 fps in the final result.

## 3 The Graphics

Mainly used [Akenine-Möller et al. 2008], slightly used [Shreiner et al. 2013].

Implemented effects

- Skybox
  The skybox was one of our the first implemented effects. We used plain OpenGL and created our own skymap, which we had to split into six faces in order to use it with the GL_TEXTURE_CUBE_MAP texture. We had to disable the depth test.

- Phong shading
  Implemented simple phong shading, with calculation of light direction and making color dependent on material constants and incident ray.

- Alpha blending
  Made alpha blending dependent on alpha channel in textures. In our case we used it for a transparent track.

- Normalmap
  Used normal mapping by creating a TBN-Matrix and transforming from texture space to world space to calculate correct normals depending on a normal map.

- Motion blur
  The motion blur effect is implemented with a technique called per-object or sometime per-pixel blur. This is realised by creating a velocity map that is computed according to the previous and current pixel in frustum space and then blurred along the velocity vector in a post process.

- Environment mapping
  Environment mapping implemented by rendering six times in six different directions to an framebuffer object and assigning the resulting images to a cube map.

- Stencil buffer
  For the portal's effect we used the stencil buffer. The idea behind this is simple: Render the whole scene once and fill the stencil buffer with zeros, except inside the portals: Here we set the stencil's value to one. In the second step we render the scene with different (higher) shader level if the value is one. In order to fill the stencil buffer correctly we used two models: One for the torus and another one for the circular plane inside the torus. These were treated as a unit.

- HUD
  On-screen text in order to visualize the current game statistics with characters rendered from a character-texture to screen by calculating the corresponding positions on the texture.

Difficulties / Problems

1. At the beginning, passing the attribute locations from the VAO to the shader were implemented wrong.
   - It follows that every FBO did not work correctly.
2. Shadow mapping (only shadow map creation works).
3. Skybox creation caused some trouble because of deprecated ACGL function for cubemap creation.
4. Could not finish tool to precompute tangent and bitangent because of incorrect calculation. Getting nan although divion by zero issues were catched.

Solutions

1. Passed attribute locations moved to draw function (quick solution but dirty, drains fbs etc). A better solution is to pass them during the shader initialisation.
2. No solution, yet.
3. Used plain OpenGL calls.
4. Computed TBN-Matrix in vertex shader and used it in fragmentshader to calculate tangent and bitangent.

## References


AKENINE-MÖLLER, T., HAINES, E., AND HOFFMAN, N. 2008. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA.

SHREINER, D., SELLERS, G., KESSENICH, J., AND LICEA-KANE, B. 2013. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. OpenGL. Pearson Education.