# Game Development as Part of the Computer Science Education

C. Rezk-Salama, S. Todt, L. Brückbauer, T. Horz, T. Knoche, B. Labitzke, M. Leidl, J. Orthmann, H. Payer, M. Piotraschke, T. Schmiade, and A. Kolb

Computer Graphics Group, University of Siegen, Germany.

**Abstract.** We report on our experience with a game project that was developed from scratch at the Computer Graphics Group at the University of Siegen, Germany. We will discuss the benefits and difficulties that arise from such a project for both the educator and the students. The aim of this paper is to clarify the aspects that must be considered to achieve game development at university level.

## 1 Introduction

Computer games have a large and continuously expanding world-wide market. Interactive digital media, games and entertainment applications are an important economical factor that cannot be easily neglected. From the point of view of a business manager, however, the development of computer games is characterized by relatively long development times and the necessary burden of considerable pre-financing with a hardly predictable outcome. These aspects put cumbersome constraints on the experimental nature of game development.

For an educator who teaches software development on the other hand, computer games are ideal examples to demonstrate the pure benefit of education. Many small game companies in countries with only weak economy have proven that almost *everybody can develop successful computer games*, provided that the necessary skills and the knowledge is available.

Another important characterization of game development nowadays is that there are little educational facilities which instruct students with the specific skills required for game development. In Germany, the above mentioned observations have given rise to facilities called *game academies*, which focus on specific education for game developers. If we thoroughly observe the current game market, however, we notice that computer games are driven to a large extent by technological innovation, such as efficient rendering algorithms and new features of modern graphics hardware. While we believe that game academies are important to provide basic skills and entry points for students focussing on game development, they cannot take over the full responsibility to convey the profound knowledge which is necessary to push forward technological innovation. We argue that such innovations are only possible with the knowledge and skills provided by high-level educational facilities, which are concerned with both research and education.

In this paper we will report on our experience with a game project that was developed from scratch at the Computer Graphics Group at the University of Siegen, Germany. We provide and discuss solutions to overcome typical difficulties that are specific to educational environments.

## 2 Why Game Development at the University?

Developing computer games at a university has several benefits and drawbacks. Maybe the most important benefit is that game programming offers a playful approach to software development from a didactic point of view. The effect is high motivation among the students, promoting their own initiative to improve their skills.

Game development is, unlike many other application areas, ideal for students to acquire *soft skills*, such as creativity and the ability to communicate their ideas with controversial discussion. We believe that this aspect originates especially from the fact, that there is no *correct* or *incorrect* solution to game play, in contrast to many other application areas.

From the technical point of view, computer games comprise a huge variety of important engineering techniques such as modular software design, efficient data structures, and interaction. Practical knowledge in such areas is important for every computer scientist, regardless of his application area. Giving students the opportunity to develop computer games as part of their education allows them to create new ideas and concepts without the danger of commercial unacceptance.

There are, however, a couple of difficulties and dangers. Apart from the above mentioned general programming techniques, there are many aspects involved in game development that are unique to computer games. One of those aspects is content creation. A university has the responsibility to convey general knowledge to maximize future career opportunities for the students. Shifting the focus too much towards one specific application area, such as content creation, is hardly justifiable.

If we neglect content creation and artwork design, however, the resulting games will soon become boring and unattractive. The main challenge for game development at the university is to find ways of realizing attractive games without focussing too much on content creation.

Besides these problems, there are many other difficulties, which all game designers have to cope with. The development of a game engine with all technical aspects can easily become very complex. However, we see such difficulties more as a challenge for software engineers, and it is the task of the educators to guide their students to meet this challenge.

## 3 A Case Study

We have started a project group to find out if game development works as educational part at university level. The project group consisted of nine students

and the development time was fixed at two semesters (1 year). At the beginning of the project the educators made a list of requirements that the project must fulfill.

- The software design for the game engine should be built from scratch. The students were allowed to use available APIs and libraries such as OpenGL, or DirectX, but should not use existing code for the engine.
- To keep the students from focussing too much on content creation, time-consuming modelling of scene geometry should be replaced by procedural geometry design.
- The game should support at least two interaction devices (gamepad, joystick).
- The game should support 3D sound.
- The project should include either artificial intelligence or multi-player capabilities.

Based on this list, we came up with the idea of a multi-player racing game with futuristic vehicles and procedural landscapes. All object motion will be controlled by physical laws such as gravitation and collision. The system will support multiplayer network games instead of an artificial intelligence subsystem.

The task of the educator in the design process is to evaluate and comment the decisions of the students, point them to possible pitfalls and sensitize them to implementation issues. The following list comprises important aspects that students are usually not aware of.

**Memory management:** Managing the available memory is crucial. Simply calling `new` and `delete` at runtime has two drawbacks. First, objects are shared between different modules and a resource manager must take care that objects are deleted if they are no longer used. The second problem is memory fragmentation. In a computer game, a huge variety of different objects, which have only a short life time, must be created on the fly. Continuously allocating and deleting small portions of memory is inefficient and will slow down the performance due to fragmentation of the memory heap. Possible solutions comprise handle-based resources [1], reference counters, smart pointers and reusable objects [2].

**Debugging:** Debugging a computer game is one of the most complicated tasks. If a multiplayer game crashes due to an error, it is impossible to restart the program and reproduce exactly the same actions that caused the error. Standard debuggers are not very helpful here, since they only provide the current function call stack. An error, however, might have happened in a completely different place and time. To be able to find errors, logging all events from the start of the game to the point when the error occurs is mandatory. Example implementations of efficient event logging and journaling systems can be found in [3] and [4].

**Profiling:** If the overall performance is not satisfactory, students must have a means to find out how much time is consumed for the execution of every part of the source code. Otherwise they will end up spending time for optimizing

code that is only executed very infrequently. If flexible and configurable frame-based profiling is required, you can implement your own profiling tool as described in [5].

**Scripting:** A considerable amount of data is required to configure each subsystem of the game. Configuration data must be tweaked frequently to adjust the behavior of each subsystem. If data is managed outside the code, modification can be applied without the necessity to recompile the game engine. The students were advised to use configuration scripts for this purpose. Available libraries such as XML-parsers or parsers based on Lex/Yacc [6] can be used.

In the design phase, after the students have familiarized themselves with the available APIs, we developed a uniform framework, which divides the complex game into separate, manageable subsystems. The functionality of each subsystem should be independent of each other as much as possible. Different subsystems communicate by exchanging events.

## 4 Implementation

In the design phase we have identified the following independent modules that our game engine should consist of. The students created UML diagrams to visualize and document the interaction between different modules before starting to write the code. Effective strategies for testing the different subsystems independently were developed.

In the beginning of the implementation, the basic framework was created including a profiler, an event manager and an object manager, as described below. Afterwards the students started the implementation of the individual subsystems.

**Graphics:** The graphics subsystem performs the 3D rendering of the entire scene graph including geometry management, frustum culling and shaders.

**Physics:** The physics module is responsible for computing the motion of every object based on input events and collision.

**Sound:** The sound module implements the playback of music, ambient sounds as well as dynamic 3D sounds caused by events.

**Network:** The network module is required for client-server communication. Different tasks must be performed during startup and runtime of the game.

**Input:** The input module translates user input from the game controllers to events used by the game engine.

The communication between the different modules is performed by sending *events*. An event can be the input from an interaction device, the collision between different objects and the like. If necessary, events must also be propagated via network to other clients.

The task of the event manager is to receive events sent by the different subsystems, maintain a log file and distribute them to other subsystems [7].

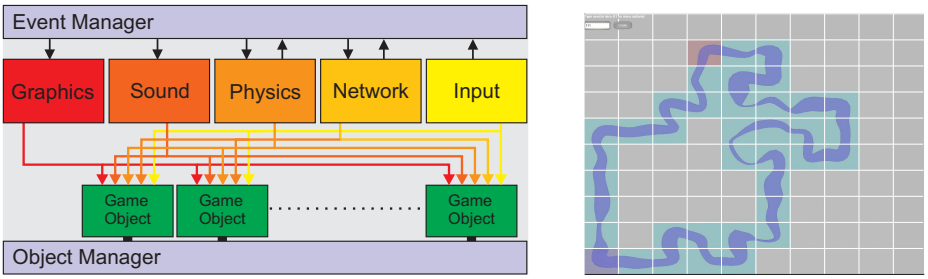


**Fig. 1.** Left: The basic architecture of our game engine. Right: The procedure that generates the race track for a given terrain is based on a uniform, rectilinear grid.

Events are triggered whenever a module requires other subsystems to react on state changes. The event manager supports an extensible set of event types.

The event manager maintains an event queue for each subsystem that needs to send or receive events. It is implemented as a singleton class [8]. The event manager must work properly in conjunction with the network subsystem, which runs in a separate thread. This requires synchronization using semaphores and critical sections to protect the event queues from concurrent access. The event manager is able to record all events with time stamps and write them to a file. This is important for debugging in order to reproduce the exact sequence of operations that lead to an invalid internal game state.

The object manager represents the run-time resource manager for the main memory. The life time of all game objects that require storage allocation is controlled by the object manager. The individual subsystems will receive notifications whenever the object manager decides to create a new game object.

All subsystems work on the same objects data. The object manager also informs other subsystems when a game object is about to be destroyed. Life time of a game object is controlled by a reference counting mechanism, which causes the object to destroy itself when the last reference is dropped. This is useful if one subsystem still requires access to the data of an object after it has been removed from the game.

### 4.1 Procedural Terrain

The basic terrain geometry was created by the use of *Fractal Brownian Motion* (FBM). FBM performs a spectral synthesis with noise functions of different frequencies and amplitudes. The geometry of the terrain is modelled as a height field obtained as a sum of 2D noise functions with a fractal power spectrum, which means that the frequency of each function is inversely proportional to its amplitude. After the generation of the basic height field, the race track is added in a separate procedure.

The terrain is overlaid by a rectilinear grid as displayed in Figure 1, right. The size of cells can be adjusted to control the granularity of the race track. A maze

algorithm is used to roughly determine the course of the race track, ensuring that the track does not intersect itself. The algorithm starts at a specified cell and chooses the next cell using the Mersenne Twister pseudo random number generator [9]. The exact course of the race track is generated by randomly placing control points for Bezier splines in the grid cells.

The final terrain geometry is created by removing the high frequencies from the height field if a vertex lies inside the track and by raising vertices below the water level. The resulting terrain is split into manageable rectangular tiles. The graphics subsystem decides at runtime which tiles must be resident in graphics memory using a simple frustum culling technique. If new tiles must be swapped in, vertex buffer data (position, normal and texture coordinates) is transferred asynchronously by the DMA controller from host memory to local video memory. In the meantime the CPU can perform physics calculations.

For texturing, a fragment shader was used to interpolate between different texture images according to the height value of the surface, resembling a natural environment, for example with green plains and snow-covered mountain tops. Texture images with different scales were applied as tiles. Due to the height field interpolation, the repetitive tiling is not noticeable from the perspective of the player. The texture images we used were obtained from free texture libraries in the internet.

Each landscape can optionally be decorated with a water surface on a specified level relative to the maximum and minimum height of the landscape. This surface is slightly moved up and down to resemble a subtle tide at the coastlines. The water surface is textured with a semitransparent decal texture. Texture coordinates are perturbed in the vertex program over time to imitate natural water movement.

### 4.2 Vehicle Models

The vehicle models were the only part of the geometry that had to be created manually using Alias Maya. For each geometric model different texture sets were created to provide a varying set of vehicles based on the same vertex-mesh as displayed in Figure 2.

The models were designed as subdivision surfaces in Maya and tessellated to polygonal meshes. The meshes were exported and converted to a binary vertex buffer format, that can directly be uploaded to GPU memory.

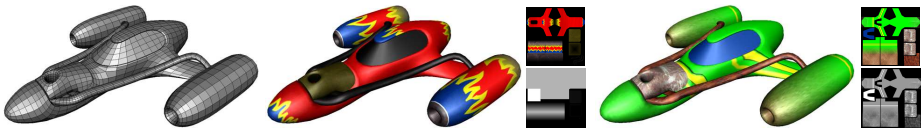

**Fig. 2.** A typical vehicle model (3750 vertices, 7232 triangles) used in our game. Each geometry is used with different texture sets.

Collisions of the vehicles with other objects should cause visible deformation of the geometry, depending on collision area and energy. The basis for this realtime deformation is a special data-format for the vehicle objects. This format provides two positions for each vertex of the vehicle: one for the undeformed default position, and one for the maximum deformation. To reduce computational load, the vehicles are subdivided into designated regions as displayed in Figure 3. As soon as a collision on one of the regions is detected, a vertex shader interpolates between the two positions of the involved vertices. The energy of the collision is taken into account as an interpolation weight. Additionally the visual appearance of the deformed regions is modified by reducing the specular term of the local illumination model according to the amount of the deformation.

### 4.3 Physics and Collision Detection

Collision detection required for our game can be divided into collisions between individual objects and collisions between objects and the terrain. Object-object collision is detected with the help of a bounding volume hierarchy, divided into three stages. A first estimate is realized by a simple bounding sphere test. The second part of the collision detection uses an oriented bounding box hierarchy. If necessary, collision detection can be performed for single triangles. We decided to base our algorithm on the OBBTree by Gottschalk et al. [10]. Since the vehicle geometry is pre-modelled, the bounding volume hierarchies can be created in a preprocess and loaded at runtime.

The intersection between the object and the procedurally generated terrain has been an additional challenge, since information about the terrain surface could not be pre-computed. We calculated a rectangle which lies directly under the vehicle during pre-computation of the bounding volume hierarchy. At runtime, we calculate intersection tests between the rectangle and the four nearest triangles of the landscape. This requires a maximum of eight tests per frame. In practice, we found that the precision of this simple rectangle test already suffices for good collision detection with the terrain.

All game objects are modelled and animated as rigid bodies according to the physical laws of motion. Every object has a set of physical properties, consisting of constants such as its mass, the moments of inertia, friction coefficients as well as variables such as position, orientation, velocity and acceleration.


**Fig. 3.** Every model has predefined deformations for different regions. At runtime the differently deformed shapes are blended together in a vertex program.

The vehicles are moved by propulsion turbines, each of which apply a certain force to a vehicle. By controlling the driving power of the turbines individually, the vehicle can be moved forwards and backwards, and also be turned around. Additional forces are caused by collisions. All forces on the vehicle are summed in order to calculate the current linear and angular acceleration. Finally, Newton's laws of motion specify the influence on the current velocity (linear and angular) as well as the current position and angular displacement.

The first tests of this model showed that a world following simple laws of kinematics was hardly playable without frictional forces. The resulting motion was similar to billiard balls rotating around its centers and flying in straight lines. The first countermeasure was to account for atmospheric drag. We also restricted the motion along the vehicle major axis and up to 15 degrees from it. This improved the handling enormously. A second modification was necessary, since the vehicles can freely rotate along all axes and they will, if they happen to collide with anything. For the player it was almost impossible to stop his vehicle from rotating with only two propulsion turbines. We solved this problem by adding a damping force which slows down the rotation if no further force or steering influences are applied. The damping force also rotates back to an upright position if angular velocity is small. Last improvements were made only by tweaking the physical parameters. The result was an intuitively steerable vehicle without feeling unrealistic. The control of the vehicles is sensitive enough to push opponents off the track, if the player decides to do so.

### 4.4 Input and Network

Modern racing games support many different input devices, such as joysticks, gamepads, racing wheels, mouses, and the keyboard. DirectX provides methods for initialization and usage of all those devices with the DirectInput part. During each frame, the current state of all relevant input devices is polled once, the data is interpreted and game engine events are generated accordingly. Furthermore, we take advantage of the DirectInput-API when accessing its force feedback interface and functionality. Since the design and usage of force feedback is complex, we only used a very basic effect for demonstration purposes.

The multiplayer functionality of our game engine is based on a client-server architecture. The server is a separate executable, which can be started by the client who initiates the multiplayer game. Network communication is performed via TCP/IP using the deprecated DirectPlay 8.0 layer.

The server process maintains a list of all players that are currently connected and assigns unique identifiers to each client. The server stores the configuration settings such as the player's names and the selected vehicle geometry as well as the seed points for the procedural map and the race track.

At startup the client process enumerates all available servers in the local area network. The user can enter the address or DNS name of the server to participate in the game. At runtime, the main task of the server is to synchronize the start of the race and to deliver incoming event packages to the client. The server program has a simple GUI, which displays status messages for debugging purposes.

**Fig. 4.** Screenshots from the game. The left image shows the water surface and the lens flare effect. The right image shows parts of the procedurally generated terrain with the race track

The task of the client network subsystem is to pack all necessary events and send it to the server as well as to unpack received network packages and propagate the event to the event manager. If a player connects to the server, notification events are used for initializing the map and for creating the necessary game objects for the new player.

### 4.5 Sound

Sounds being used in computer games can be separated into three different groups: Music being played for entertainment purposes mainly, static sounds providing acoustical information and dynamic 3D sounds.

Static sound elements such as menu sounds (e.g. an audible click when pressing a button), network event sounds (a "pling" when a connection between hosts has been established successfully) or in-game event sounds (an audible notion when a player picks up a power-up) do not need to be adjusted dynamically with the game flow. They sound the same whenever they are played and can thus be associated with a certain event. Further static sounds are ambient ones being played during game play to enhance the game atmosphere.

Dynamic 3D-sounds are the most important sound elements within a game. They contribute to the realism of the sound rendering to a great extent. To exploit the benefits of dynamic 3D sound rendering the sound render engine has to be compliant to the *Interactive 3D Audio Rendering Guideline (I3DL2)*[11]. With DirectSound, a sound programming interface fully embodying I3DL2, a minimum of 32 sounds can be played simultaneously taking into account the listener's and sound object's position, orientation and velocity as well as the sound's radiation pattern. Further effects applied to the sounds like reverberation, attenuation or Doppler effect are dynamically adjusted.

A very difficult part in 3D sound programming is debugging. At this stage, students will need a silent room with a surround sound system, and a lot of time

for testing. Concentration and patience is required to determine whether the visual and acoustic impressions are corresponding or not. Evaluating variances is even harder due to the fact that the minimal audible discrepancy is as low as two to ten degrees in the horizontal plane and nine to 22 degrees in the vertical plane [12].

The situation gets even worse if the graphics of the game cannot be used for debugging yet. For this reason it is strongly recommended that students develop an easy to handle test program at an early stage of the project. It should include a correct visual representation of sound sources and be independent of the rest of the game. This can save a lot of time and effort.

Audio content creation for games, let alone composing music, is demanding but essential. Sound and music have a great impact on games, unfortunately this is only recognized when a game lacks good music and sound. Although in a university project this aspect seems to be less important than in commercial projects, it should neither be neglected nor underrated. Students who take over the sound part of a game project should know that the content creation takes at least the same time as the programming. It is difficult to find good and suitable sound files for effects, collisions or propulsion sounds. As a result, some of the game sounds in our game were produced by recording the sound of a vacuum cleaner and alienating it until it sounded like rocket propulsion. Though that might be the same approach as that of professional sound artists, it is extremely time-consuming if students have no experience in professional sound production and the usage of professional sound equipment.

### 4.6 Special Effects

After the different modules have been implemented and the complete system has been assembled and tested, there was time left to enhance the visual appearance of the game by including visual effects and additional features.

Particle systems are an integral components of recent games. Several visual effects such as smoke, fire, rain, snow, and explosions, can be realized by the use of particle systems. We implemented a stateless particle engine as an independent subsystem that runs entirely on the graphics board. It offers the basic functionality for a variety of visual effects like the above mentioned. The system renders individual particles as point sprites. The texture of the point sprites, coloring, direction and size of the particle system can be adapted to the required needs to realize different effects.

The particle movement is implemented as a vertex shader on the graphics processor, which calculates the motion of each particle as a function of its initial position and velocity and the current time. The graphics processor calculates the animation of a large amount of particles without affecting the frame rate significantly.

A lens flare effect is an image artifact caused by the lenses of a camera when facing a bright light (Figure 4, left). We produced the lens flare effect by using a set of semi-transparent texture images which were applied to screen-aligned billboards. The billboards were arranged on a vector from the position of the

light source to the middle of the image plane. To determine whether or not the player is looking into the light, the position of the light source must be culled against the viewing frustum.

A heads-up display (Figure 4) provides the player with information, such as speed, round times, current car position and more. A mini map of the terrain is created by rendering the entire procedural geometry into a texture image. The position on the mini map and its rotation are determined by the vehicle's position and orientation. Similar to the mini map, we implemented a rear-view mirror rendering the scene with an opposed camera direction. The frustum culling technique in the graphics subsystem must assure that the required tiles of the terrain are resident in local video memory as well.

## 5 Conclusion

Developing an ambitious computer game in an educational environment is a challenge for both the educators and the students. For the educators, the described game project was an experiment to evaluate whether or not game development works as part of the computer science education. Although the result is still far from being a commercial-level engine, the project already contains all important aspects of a professional computer game from the technical point of view.

Creating a structured and detailed software design is essential. The educator's task in this design phase is to keep all students involved. In some cases the educators will have to slow down some of the students, who would rather like to start writing the code. Other students seemed to be overwhelmed by the complexity. It is important for the educator to keep all students motivated, and this requires discipline among all students to plan the project thoroughly before writing the first line of code. In the design phase the educator should advise the students which parts of the architecture must be fixed at the beginning and which decisions can be postponed and discussed later. Before starting the implementation, everybody must know exactly what to do.

Since the technical aspects of game development are in focus, the educator should take care that not too much time is spent for content creation. Although we were aware of that problem and tried to solve it using procedural graphics, the time consumed for content creation, such as geometry and sound was still underestimated.

Writing the code to implement the different modules probably was the least time-consuming task. Most of the time was consumed for understanding and evaluating poorly documented APIs, for ensuring that the different components work together as expected, as well as for debugging, parameter tweaking and performance optimization.

After finishing the project, the students stated that, if they had to start over again with the project, they would definitely revise one design decision or the other. This is not unusual. Above all, it approves that the students have expanded their skills and gained experience with the large and complex project. They have become familiar with project management as well as practical tasks

such as integrating and enhancing third-party code, which as well may be poorly documented and contain errors. This experience obviously will be of great value for any type of software project that they might work on in the future.

At the bottom line, we find that the game project was a full success. In the end, both the educators and the students were somewhat amazed about the complexity and the quality of the final result, although there is always room for improvement.

## References


1. Bilas, S.: A Generic Handle-Based Resource Manager. In DeLoura, M., ed.: Game Programming Gems. Charles River Media (2000) 68–79
2. Boer, J.: Resource and Memory Management. In DeLoura, M., ed.: Game Programming Gems. Charles River Media (2000) 80–87
3. Hawkins, B.: Lightweight, Policy-based Logging. In Treglia, D., ed.: Game Programming Gems 3. Charles River Media (2002) 129–135
4. Robert, E.: Journaling Services. In Treglia, D., ed.: Game Programming Gems 3. Charles River Media (2002) 136–145
5. Evertt, J.: A Built-in Game Profiling Module. In DeLoura, M., ed.: Game Programming Gems 2. Charles River Media (2001) 74–79
6. Kelly, P.: Using Lex and Yacc To Parse Custom Data Files. In Treglia, D., ed.: Game Programming Gems 3. Charles River Media (2002) 83–91
7. Harvey, M., Marshall, C.: Scheduling Game Events. In Treglia, D., ed.: Game Programming Gems 3. Charles River Media (2002) 5–14
8. Bilas, S.: An Automatic Singleton Utility. In DeLoura, M., ed.: Game Programming Gems. Charles River Media (2000) 36–40
9. Matsumoto, M., Nishimura, T.: Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudorandom Number Generator. ACM Trans. on Modeling and Computer Simulations (1998)
10. Gottschalk, S., Lin, M.C., Manocha, D.: OBBTree: A Hierarchical Structure for Rapid Interference Detection. Computer Graphics **30**(Annual Conference Series) (1996) 171–180
11. MIDI Manufacturers Association Incorporated: IASIG : IASIG Interactive 3D Audio Rendering Guidelines (Level 2) (1999)
12. Pulkki, V.: Spatial Sound Generation and Perception by Amplitude Panning Techniques. Espoo **62** (2001) 8