Developing a Jump'n'Run Puzzle Game

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

Samiro Discher* Alexander Kugler† Christopher Kugler‡ Christof Mroz§

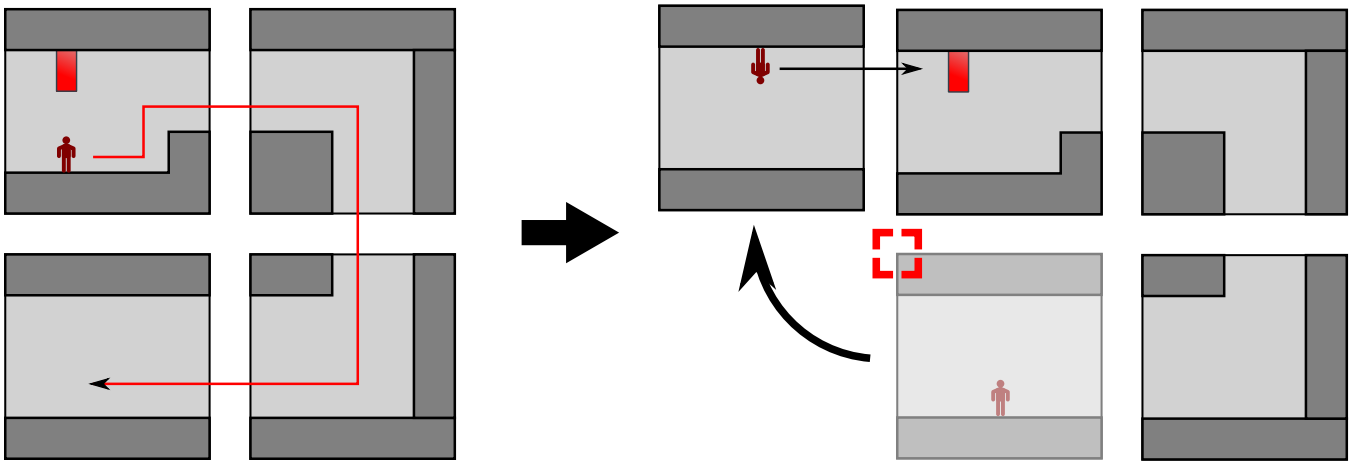

Figure 1: *Solving a basic level, composed of four rooms.* Note: *Gaps between rooms are illustrated for emphasis only and don't appear in the actual game.* (Left) *The player character is subject to gravity, and thus can't reach the red exit on the ceiling as-is.* (Right) *By walking to the lower left room and flipping it along its upper left corner, the exit finally becomes accessible. This is possible because gravity always reorients according to the main character, as per the game rules.*

Abstract

The objective was to develop a side-scrolling jump'n'run with puzzle elements, where heavy emphasis is placed on a modern presentation employing state of the art 3D graphics effects and filters.

Each level is divided into distinct, quadratic rooms arranged on a grid, as exemplified in Figure 1. Any time during gameplay, the room containing the main character can be rotated (subject to some rules, of course) in order to rearrange its grid position and orientation, so that what was previously a ceiling or wall can suddenly become a floor with respect to adjacent rooms, and vice versa.

We decided to use a deferred rendering pipeline due to stylistic decisions. For example, the game takes place in an abandoned space ship, i.e. a sci-fi setting, and consequently draws a lot of its atmosphere from lighting. Deferred lighting makes handling of many light sources comparatively easy. Conversely, well-known disadvantages of deferred rendering, like transparent objects, proved to be unproblematic due to the predictable 2.5D camera angle.

Keywords: game programming, jump'n'run puzzle game, deferred rendering

1 Gameplay

Game mechanics were largely inspired by the indie puzzle jump'n'run *Continuity* [Lima and Mikaelsson 2010]. While we enjoyed the basic idea, the low difficutly disappointed us on other hand. Ultimately, only the core principles remained to create what we hope results a more challenging experience. Refer to Figure 1 for an overview of the resulting rules.

The game state can be in either of two modes, namely *action mode* (default) or *meta mode*. In action mode, the main character can be controlled just like one would expect it from any other jump'n'run in terms of controls and physics. At any time, *meta mode* can be entered. The simulation stops and the camera zooms out to give an overview of the level. Then, the player can select one of the current room's corners to *rotate* this room around it. Here, the *current room* refers to the one containing the main character.

Rotation alters orientation of a room. Because gravity also always follows the main character's orientation, this allows one to walk on surfaces that were previously ceilings or walls. Thus the player reaches seemingly inaccessible areas, defeats enemies (note that the character is unarmed) and solves other such puzzle-related tasks. Any room that may look useless, at first glance, could be repurposed in unexpected ways if considered from right perspective. While a closed door is usually a bad thing, when rotated it can be used as a bridge (which, to the contrary, is of course ideally not open), and so on. Besides, the rooms themselves contain usual platformer puzzle elements like switches. These sometimes also operate on adjacent rooms to create even more complex dynamics.

2 Rendering

Modern, high quality visual representation was by far the highest priority goal of the practical course right from the beginning. With this in mind, and despite the predominantly 2D game logic, we settled on a deferred 3D renderer (without light pre-pass).

A move which was also motivated by stylistic choices. The game is set in an abandoned spaceship, i.e. a gloomy, industrial sci-fi scenario. Consequently, liberal number and placement of light sources

*samiro.discher@rwth-aachen.de
†alexander.kugler@rwth-aachen.de
‡christopher.kugler@rwth-aachen.de
§christof.mroz@rwth-aachen.de

is crucial to create the right atmosphere. Deferred lighting is a comparatively straight forward solution to this problem that scales well at the same time. Shadow mapping was also implemented and used occasionally to place accents, along with screen-space glow. Further, we targetted a slighty stylized but still realistic look and feel. Screen-space effects like SSAO and motion blur boost believability a lot with only constant costs, in this regard.

Deferred renderers are usually at a disadvantage when dealing with transparent objects. This turned out not to be an issue in our specific case however, since the 2.5D side-scrolling camera angle made Z-order very predictable. A second pass with only transparent geometry is all it takes. Real-time refraction is achieved simply by sampling the opaque pass' result along a direction depending on the pixel's screen-space normal. Recent advances in screen-space anti-aliasing, like FXAA [Lottes 2011], offer surprisingly good results, especially when parameters can be tuned to a certain Z range (which, again, is mostly fixed in side scrolling games).

Geometry shaders support texture, environment and normal mapping, among others. To overcome combinatorial explosion, and since we aimed for a single geometry pass for opaque and transparent objects each, we originally planned to create a master shader with tagged code regions that could be re-assembled by a script to yield the needed combinations. This idea was scrapped due to time constraints, though.

3 Tools and Asset Pipeline

We developed plugins for the freely available 3D modeller *Blender*, allowing us to model, place and export assets from the same application. Unfortunately, there is not always a sensible mapping from Blender to game engine material parameters: The former knows about glow intensity but not about color, for example (which is important for our futuristic setting). We use name mangling prefixes and custom properties to solve such ambiguities.

Bone based IK animation for characters can be *baked* by sampling the Blender IK solver's output at keyframes. More than one such animation can be associated per character. At runtime, keyframes are interpolated via SLERP, and multiple animations are mixed via weighted NLERP to ensure smooth transitions and, while not used in our final revision, to compose distinct animations for e.g. torso and legs.

Besides assets, we can export *levels* via a designated plugin as well. While it seems like a good idea at first, re-purposing an existing tool raises a key trade-off: create a full-fledged, fully integrated plugin with knowlege of game engine semantics or rather rely on hacks and conventions to save time? We chose the latter. Hacks are acceptable for us as long as they are documented clearly.

In reality though, documentation got out of sync quickly, and eventually the code became the documentation. Hence the artist needed to understand the exporter code (or, of course, ask the developer whenever something breaks), which is not a reasonable requirement and was not the case in our team either. Without doubt, completely custom tools (even a simple XML description of levels), or at least more effort spent on UI, could have saved us time in the long run.

4 Software Engineering

Neither of us wrote a complete game before (well, besides *Tetris* perhaps), so maybe it's no surprise that the engine didn't turn out too clean; to the point that we started to rewrite the engine from scratch. Despite much higher productivity at the second attempt (and, admittedly, more fun due to less boilerplate and toy block like modularity) we were not able to complete it on time, however.

Challenges were frequently underestimated because our idea of a 2D jump'n'run is strongly influenced by classics like *Mario Bros.* and *Megaman*, and our game logic is not really more complex if one ignores presentation. When striving to create a lively, interactive and believable world, however, which in turn requires lots of animations, feedback and dynamics as well as variable time step, presentation logic starts to creep into game logic quickly and complicates matters.

We adopted an object oriented MVC approach suggested by our supervisors. In classic MVC the Controller reads and writes to the model, which is then read by the renderer, directly. The renderer needs deep knowledge about the model, which was not acceptable in our case: the graphics group was burdened by learning about graphics programming enough already, and had no time to learn C++ and keep up with the game logic. Instead, an extra *presenter* layer reorganizes the model into flat lists of *scene nodes* first, which are then interpreted in a *dumb terminal* fashion. This turned out to be the only design decision we got right from the start.

For the first engine, we coerced entity types into an OO class hierarchy. Despite clear requirements from the beginning, more and more features were moved upwards in the hierarchy to speed up development time (*blob* anti-pattern [West 2007]), or simply because it was not clear where to place it. What is the solution? Think harder?

Maybe, but we ditched OO hierarchies altogether instead. The new engine was rather based on a true entity system, i.e. according to the original meaning of the term from database theory [Martin 2007]. In OOP terms, entities can be vaguely described as mere containers for collections of data called *components*. Distinct types are not pre-declared like in OOP but it is rather possible to *query* entities for certain abilities on demand. For example, a *door* is described in terms of its components as

$$\begin{aligned}\text{Door} \;=\; &\text{2D Coord} \;+\; \text{Door logic} \;+ \\ &\text{Rigid body} \;+\; \text{Asset} \;+\; \text{Animation} \;+ \cdots\end{aligned}$$

5 Conclusion

Recombining existing level parts to form new layouts is a promising and fun concept that deserves further experimentation. Side-scrolling games are agnostic to some common drawbacks of deferred renderers. Tools should either be fully developed plugins, actively imposing game engine semantics into their host applications, or completely custom to begin with. In a graphics-heavy game *much* complexity stems from presentation logic, compared to "actual" game logic complexity. It must therefore be carefully incorporated into design stages from the beginning. Managing game objects in an entity system fosters creativity, while OOP inhibits it due to costly refactoring.

References

LIMA, G., AND MIKAELSSON, S., 2010. Continuity - a sliding-tile puzzle game. `http://continuitygame.com/`.

LOTTES, T. 2011. Fxaa. *NVIDIA White Paper*.

MARTIN, A., 2007. Entity systems are the future of mmog development. `http://t-machine.org`.

WEST, M., 2007. Evolve your hierarchy. `http://cowboyprogramming.com`.