Technical Design Document

Robert Cupisz | Lead Programmer, Team 4
DADIU March production 2009

1 Overview

The game is of quite well explored 3D action puzzle genre – this time in a *casual* flavour. It shares a lot of game elements with it's predecessors, thus a lot of problems the production is facing have already been solved and the solutions can be reused.

These standard elements are: roughly box-shaped, interconnected rooms

filled with characters moving about in a traditional fashion, doors, lifts, moving platforms, floor buttons, etc.

The twist, however, lies in artificial intelligence and graphic effects. AI has to be quite elaborate to give the player satisfying interaction with the numerous NPCs. Top-notch graphic effects help the visual design and are primary means of rewarding the player.

2 Short note on technology

Unity game engine has been chosen for the production specifically due to it's:

- extra short code-test cycle
- documentation
- good debugging feedback
- good game objects scripting interface
- bug-free physics engine
- smooth asset pipeline
- good shader authoring support (also extra short code-test cycle here)
- web deployment possibilities

None of the above can be said about the competing products such as Source game engine.

Spikes

A *spike* in the Agile Software Development terminology is an experiment that allows developers to learn just enough about something unknown in a user story, e.g. a new technology, to be able to estimate feasibility and time needed for implementation.

All potentially problematic areas had been *spiked* on the target platform either during preproduction or in the early phase of the production. These areas included for instance: asset pipeline, handful of graphic effects and use of AI frameworks.

3 Game elements

Following is a discussion for the strategy for implementing all the game elements together with risk assessment for each for each of those.

Game controls – prototyping

One of the biggest concerns in the early days of the production was the possibility of creating controls, which would be easy to immediately grasp by a casual gamer, yet offering seamless interaction, so that the user could express himself freely within the game world.

A high-placed top-view camera was an obvious choice for the selected target audience, as first person perspective would make it harder for a non-gamer to understand the spatial organisation of the surroundings.

For the character controls, however, three competing prototypes has been created very early. This allowed for testing both which of those schemes works best for the target players and also if the core game mechanic is easy to understand.

The tests based on prototypes gave very valuable results: we have settled for a mouse-only control scheme and have learned, that the mind-controlling mechanic is too difficult to understand without presenting proper feedback for the player's actions. We have also learned, that the gameplay idea was considered to be interesting and fun, which took a lot of risk off the whole production.

Overall it's hard to overestimate the value of prototyping.

Puzzle building blocks

Building a lot of varying puzzles requires numerous puzzle elements, each of which with it's own and easily customisable behaviour.

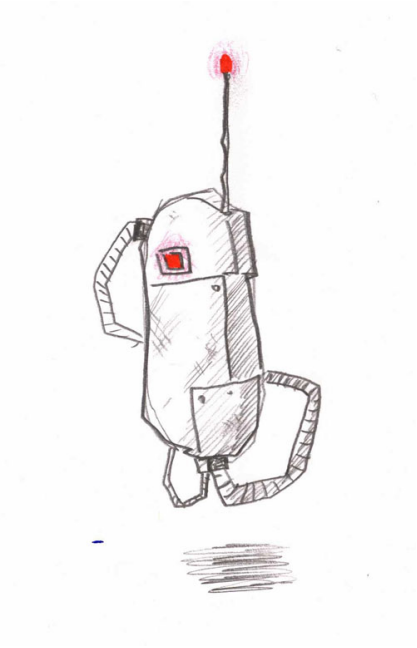
Thankfully Unity exposes a well-designed interface to do just that: a puzzle element is being created and it's behaviours scripted by the programmers; the whole object gets sealed in the *prefab* package, which is reusable by the game/level designer and allows for customisation via public variables without programmer intervention.

This task was in fact so easy, that it was possible to quickly create many draft puzzle elements, which lead to early creation of the first levels. All in all the risk of this job was estimated as very low.

Artificial intelligence

Programming AI for the game means scripting behaviour for the robot and slave characters and solving the pathfinding problem.

As both those tasks are important for the core gameplay and user experience, we have conducted a test of the available frameworks in the early production phase. These frameworks are correspondingly: Behave and Path, created by the user AngryAnt to be used specifically with Unity.



Behave is using the industry-standard concept of behaviour trees to combine and organise simple behaviours to a structure capable of reacting in a very complex way. Path is as standard and uses navigation meshes to solve the pathfinding problem.

Both those frameworks are in an early beta state and currently under development, as the new version is to be released before the end of March. Knowing that we were prepared to encounter bugs and even our early tests detected some of them.

As we knew however, that no other frameworks tackling those problems are available, and implementing our own would take weeks, we have decided to use AngryAnt's implementation. As this undeniably results in putting the production at high risk, we have prepared easy to implement alternative solutions of lower quality for each of the AI aspects.

For instance instead of having the slaves automatically recalculate it's patrolling path, when a door seals off it's current path, we will have a custom script detecting the door being closed and re-routing the slave to a pre-defined alternative route.

Graphic effects

Graphic effects such as custom shaders, particle effects, etc. are important to the game in the sense, that they aid the visual design to pop-up and serve as a reward in itself, which can be carefully dosed by the game designer to keep the player interested in the game.

Although some of the planned graphic effects are very difficult to implement and pose a high risk of failure, they are never a threat to the production itself. If we fail to implement given effect, there's always possibility to creating a similar effect in a different way or resigning from it entirely to create different eye-candy somewhere else.

Even while most of the problems with visual effects could be solved that way, we still wanted to conduct two *spikes* for the two most important graphic effects in the game. The spikes for the avatar and the healing effect ended up successfully, which leads me to conclusion, that graphic effects failures will not be a threat to the production.

The performance aspect will be discussed in the last section.

Level management

The game is divided into levels, levels into sections and sections into individual rooms or puzzles.

Each section ends with a checkpoint. When the player fails in the game, he's being respawned at the last checkpoint and puzzles after that checkpoint are being reset. This approach is quite tedious to implement, but is of little risk.

Each level ends with a loading screen and a new level is loaded. That solution is preferable when dealing with web player, as the game can be started immediately after the first level finishes downloading (subsequent levels are streamed in – in the background).

Having a seamless transition between the levels (separate floating islands in the same scene?) would result in a better experience, but is more technically challenging and was moved to a *nice-to-have* list.

Assets

Before the production an extensive description of rules for creating assets has been created, which allowed us to avoid some of the common pitfalls.

As soon as an initial version for each of the more problematic assets (rigs, animations) is created, it is being put into the game editor to test if it imports and works correctly.

Another issue is in putting the immense amount of assets into the game, as it needs to be done by at least a technically savvy person. Fortunately,

apart from the programmers, our Game Designer has mastered the Unity editor, which makes us have enough hands for the job.

4 Performance optimisation

The final stage of the production will be focused on debugging and optimising performance of the software.

The proper approach to optimisation is firstly using a set of standard techniques for identifying bottlenecks, which might be one of the four kinds:

fill rate issues may be dealt with lowering the amount of work done per pixel, so simplifying the lighting model and the effects in shaders in general; checking for excessive overdraw can also help;

triangle rate problems are solved by lowering triangle resolution of some of the game objects, decreasing number of props or combining separate meshes to reduce number of draw calls;

pathfinding – checking for excessive recalculation of paths and lowering navigation mesh resolution;

physics simulation performance problems might be caused by mesh colliders of too high resolution, long chains of interconnected rigidbodies or generally too heavy computations in the FixedUpdate() methods.

5 Division of labour

Since Unity's scripting interface allows for quick implementation of typical gameplay elements, the job can be handled entirely by one programmer.

This allows us to have the two other programmers focus correspondingly on AI and graphic effects – making them a lot more effective in their work.

At the same time each of the programmers keeps track of current state of the code base and solutions used by his team-mates, making it possible for him to aid/replace the other programmer in his specialised area. This is important to keep the project's *truck factor* at a reasonably high level.

6 Conclusion

All in all, the needed preparation tasks had been performed in time and it is safe to assume, that the team is set-up as well as possible for the short and inherently risky one-month DADIU production.