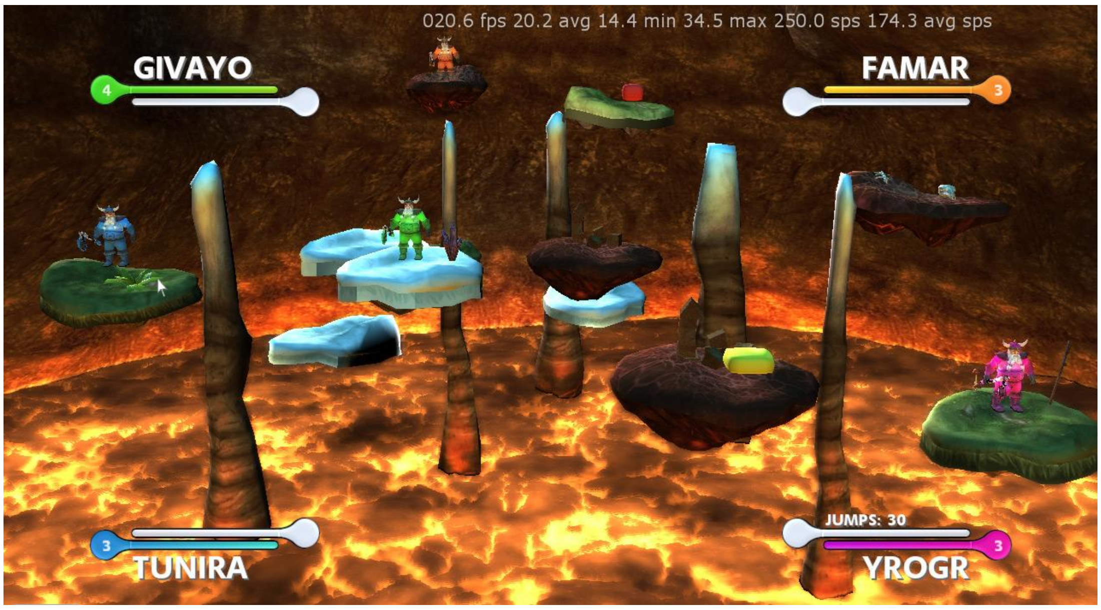
PART 4 – ALPHA RELEASE

THE PRODUCT

The current release features all items from the desirable target. Graphics-wise those are: Textured pillar and island models, textured cave, animated lava and a generic particle system used for the ice spike, the flamethrower and explosions. Furthermore there is a newly designed HUD and a menu screen. On the game play side we have moving islands, island attraction, island repulsion, ice spikes, the flamethrower and direct combat in the form of hits with pushback. HDR from the high target is also already implemented.



020.6 fps 20.2 avg 14.4 min 34.5 max 250.0 sps 174.3 avg sps

CHANGES

Statistics (ReqG06) and high score (ReqUI02) have been moved into the high target. The default means of moving among the island has been changed from island jump (ReqP06) to island attraction (ReqP04). The jet pack can only be used when falling as a mean of saving one selves and doesn't use any more fuel, as this has been removed as a resource. Island jumps have been restricted to certain distances. Island walking (ReqP05) has been replaced by island jumps after attraction. Island jumps over long distances and island repulsions are now only available through power ups. As a result of some testing, we streamlined the controls and put all means to move between islands (jump, attraction and jet pack) on one button – this will be evaluated further in the play testing phase. Furthermore, to improve performance we split rendering and simulation into two different threads.

ACHIEVEMENTS

Many features have been polished since the last milestone and are now visually pleasing and more usable. Those include:

- The lava effect (ReqL03) has been merged into the game, optimized and polished.

- Pillar and island models have been improved and textured. There are three island styles (burnt, icy and green) with different decoration.
- Power ups and selection arrows bounce in a sinus wave, so they can be spotted more easily
- The HUD has been completely redesigned and line with the streamlining of game play (only two bars: health and energy, indication of jumps and island repulsions).
- We created a particle system (see production examples) which has been used for the ice spike, and will soon be implemented in the form of explosion and burning effects.
- The ice spike's aiming has been improved, it is now able to avoid islands and pillars to a certain degree to better hit its target.
- Islands which get attracted push away other islands in their path and quickly arrive at their destination, though deceleration slowly towards it.
- The general movement of islands has been improved and hovering back to their original path around a pillar has been implemented.
- The implementation of a multithreaded architecture (one thread for rendering, multiple for simulation and collision detection) resulted in a large performance boost (see production examples).

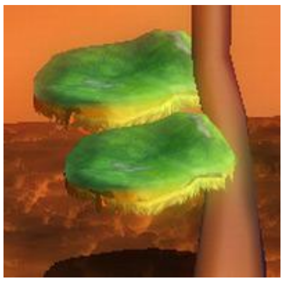
PROBLEMS

Most of the bugs should be fixed until the alpha presentation of Tuesday and the play testing of this week. We track all existing bugs using our own Mediawiki. Our biggest problems right now are keeping the performance around 30fps and eliminating the jerkiness which can result from the decoupling of rendering and simulation.

PRODUCTION EXAMPLES

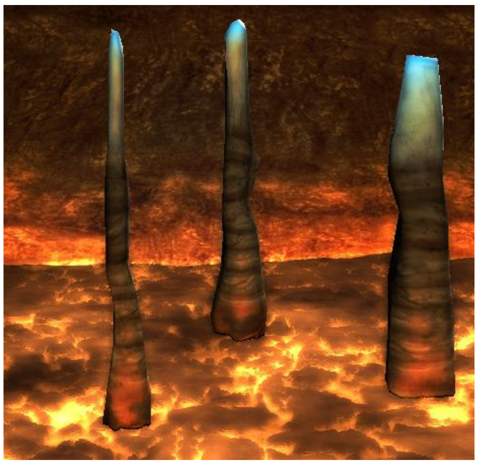
ENVIRONMENT

The environment now features different elements in their final version which previously were only available as dummies:

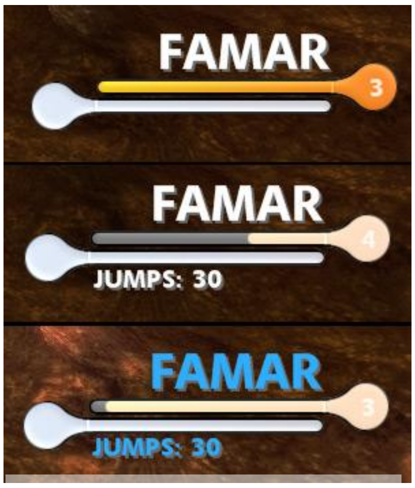

2 islands next to a pillar.

First, we have different island models now which correspond to different altitudes inside the cave: There are grassy islands on a middle level, icy islands on the upper and burnt islands on the lower level. Using pixel shaders, we added some windy grass parts on the side of the grass islands. Also, we have new models for pillars which feature snow on the top to indicate that they are on a higher altitude. As for the lava, we decided to go for a variation of the third picture in the interim report chapter. The cave can be seen in the background as well.

All elements except the lava are shaded with a fully-functional Phong shader with individual extensions, lit by three directional light sources. The first light source is a warm orange from below to simulate elements lit by the lava. This one has a very fast decay as can be seen on the image below. The second light is a cold blue one from above, indicating daylight. By tuning these lights accordingly, we've been contributing to the contrast between cold and warm tones which we wanted to achieve right from the start. The third light source, finally, is a moving spotlight coming from behind. This emphasizes the feeling that the actors are actually fighting in an arena and increases the contrast in the border regions. It will be a new high target to add a moving spot light model to the cave wall to justify this particular light source.


Three pillars in the lava inside the cave.

HUD



The new HUD. Top: Full Health and Energy. Middle: Damage is being sustained at the moment. Bottom: Player is frozen.

We created a new version of the HUD which has a focus on simplicity and better visual integration into the whole scene. Also, we added some blinking effects whenever damage is sustained in order to increase the feeling of being hurt (analogously to red tinted screens in shooting games). When the player is frozen, his name blinks in blue tones. The number of lives is displayed as a number inside the health bar. Below the energy bar, the current power-up details (if any) are displayed.

The HUD is mirrored according to its position on the screen – this can be seen in the full-screen picture above. In terms of implementation, the bars consist of different monochromatic components which are colored and combined in a pixel shader.

ANIMATIONS

The animated dwarf character standing on an island.

We are using the XNA Animation Component Library (ACL) from http://www.codeplex.com/animationcomponents to animate the player characters. We had to change some parts of the library since our models now need to be processed by both our own processor as well as the ACL processor. As for models, we are currently using the standard dwarf model that is supplied with the library but we hope to get an own model into our game within the next weeks. In terms of coloring the players, we have an alpha map on the dwarf's texture which indicates which parts of the model may be colorized how to what extent.

MULTITHREADING

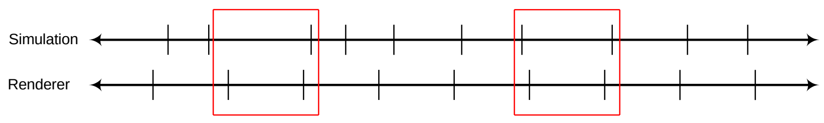
OVERVIEW

Running Project Magma on only one of the three cores of the Xbox resulted in performance problems during the alpha phase. The obvious optimization was to divide the game into a rendering and a simulation thread since both used about the same amount of CPU time.

One suggested approach was not to care about synchronizing the two threads, but to just use the simulation data as-is inside the renderer. But this approach could lead to inconsistent frames and possibly some other more serious problems. Another possibility is to keep the data the renderer needs in two places. On one hand the simulation still owns the master data; on the other hand the renderer owns a copy of the data which is periodically updated. Using this concept synchronization can be achieved with a minimal amount of overhead.

TECHNICAL DETAILS

The technical realization of the mentioned approach is quite trivial. On the simulation side a set of renderer-stubs are installed. The simulation always updates the data that needs to be passed to the renderer on these stubs. These stubs record changes using change-objects in one queue per simulation frame. After the simulation frame is done some intermediate code passes the queue in a synchronized manner from the simulation to the renderer. Whenever the renderer wants starts to render a new frame it checks whether it has received new update-queues. If so, it applies the changes provided by the queues to its copies of the simulation-data. This results in the renderer being updated in a consistent manner with only one short point of synchronization, namely the passing of the update queue. In addition the creation and application of the update queues is usually not a huge performance hit since there are not that many changes due to temporal and local coherence. The memory overhead is also quite low because the renderer does not need that much data to be copied. Usually it only needs copies of position, rotation and scale.



The drawback of this approach is shown in the illustration on the right side. It visualizes the timeline of simulation and renderer frames. On each frame-change of the simulation (denoted with a vertical marker) some updates are passed to the renderer. If the simulation is running on an unstable, varying frame rate the situation might occur that the renderer has to render two consecutive frames using the same data. This can lead to a jerky frame rate which can result in the game appearing to run at a much lower frame rate. If, for instance, the renderer runs usually at about 30 frames per second the resulting sequence of pictures may look like the generated by a renderer running at half the speed (since some frames are really the same). We think that the solution to this problem lies in interpolating the values delivered by the simulation. Using interpolation we can avoid having to render the exact same frame twice, which should lead to a smooth animation of all the objects within the game.

PARTICLES

OVERVIEW

The simulation and rendering of particles is a non-trivial problem. Especially the simulation of many thousands of particles on a computer's or console's CPU is not feasible because they are just not designed for such tasks. Thus some research was used to implement the particle system for Project Magma. The main sources of information are:

- "Building a Million-Particle System" by Lutz Latta, published on Gamasutra in 2004: http://www.gamasutra.com/view/feature/2122/building_a_millionparticle_system.php [1]
- "UberFlow: A GPU-Based Particle Engine" by Peter Kipfer, Mark Segal, Rüdiger Westermann, published in 2004: http://ati.amd.com/developer/Eurographics/Kipfer04_UberFlow_eghw.pdf [2]
- XNA Sample: http://creators.xna.com/en-US/sample/particle3d [3]

There is also another interesting read that is, unfortunately, of no use, since it is using the native Xbox 360 SDK and not XNA:

- "Particle System Simulation and Rendering on the Xbox 360 GPU" by Sebastian Sylvan, published in 2007: http://www.ce.chalmers.se/~uffe/xjobb/ParticleSystemSimulationAndRenderingOnTheXbox360GPU.pdf [4]

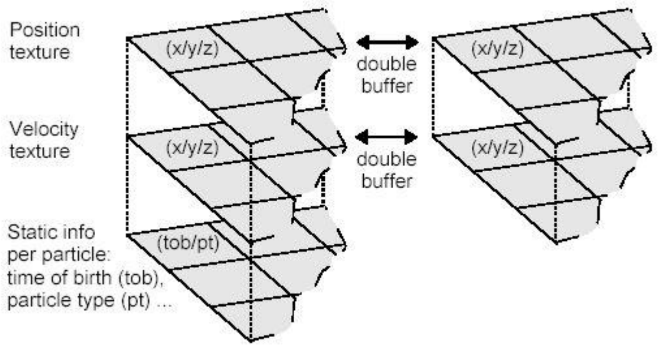
In general the readings discuss particle systems that are simulated on the GPU. But there are also two different ideas on how to approach the problem.

The first one uses so called stateless particles which are used by [3]. In this approach the programmer chooses a set of closed form functions that compute the parameters of a particle (such as position, size, etc.) depending only on a time parameter. There are several drawbacks: the particle cannot react to changing external forces and particles must always be rendered using an ever changing vertex buffer which puts a lot of strain on the bus transferring data from the CPU's local memory to the GPU's local memory.

The second approach which is taken by [1] and [2] are so called stateful particles. In this approach the state of a particle is stored inside a set of textures on the GPU. The particle simulation is then run on the GPU using a pixel shader. Using this design has the advantage that the data resides at all time on the GPU which prevents expensive transfer operations. On the other side there is the drawback that the GPU always simulates a number of particles depending on the texture size. Computational cost is therefore not determined by the actual number of particles simulated but by the maximum number of particles the system allows.

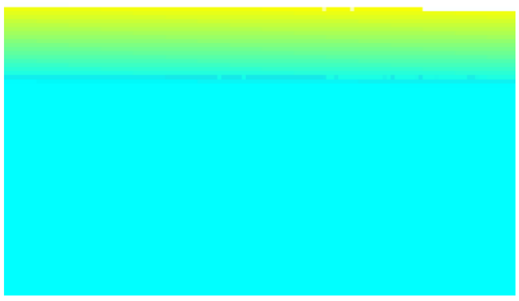
Project Magma implements stateful particles.

TECHNICAL DETAILS



When using a stateful approach to calculate a particle on the GPU the state of a particle is stored inside some resource residing on the GPU. On modern DirectX 10 compliant hardware there is the option to store particle data inside vertex buffers, process them using vertex shaders and to use the stream out feature to write to new vertex buffers. Unfortunately the Xbox does not support this feature. Therefore the state of particles is stored inside a set of textures: One position texture, one velocity texture, and a set of textures storing static particle data like its time of birth, its type, some random numbers associated to it or other data. The data is then processed by a pixel shader which outputs new values for position and velocity. Since a pixel shader cannot write into the same texture it is reading from the system needs to double buffer the position and the velocity texture. The illustration on the right is copied from the Gamasutra article and shows this setup. There is also another illustration showing the position texture of a particle system.

Creating new particles is quite easy. The systems can either track particle lifetime on the GPU, or, like it is implemented in Project Magma, use the texture as a ring buffer. New particle values are then simply rendered as single points onto their position in the texture.



Rendering is implemented using point sprites and one static, huge vertex buffer rendering each particle of the texture. Particle position and life are read inside the vertex shader using the vertex textures feature of shader model three. If the particle is not alive anymore it is offset to some off-screen location. Otherwise the shader renders it to its current position.

Running the particle system on the Xbox GPU and an additional set of modern high end GPU's has shown that the system is easily able to render many thousands of particles without noticeable impact on the performance of the game. We also experienced that currently the most expensive operation seems to be the upload of new particles to the GPU since this involves the modification of a vertex buffer.