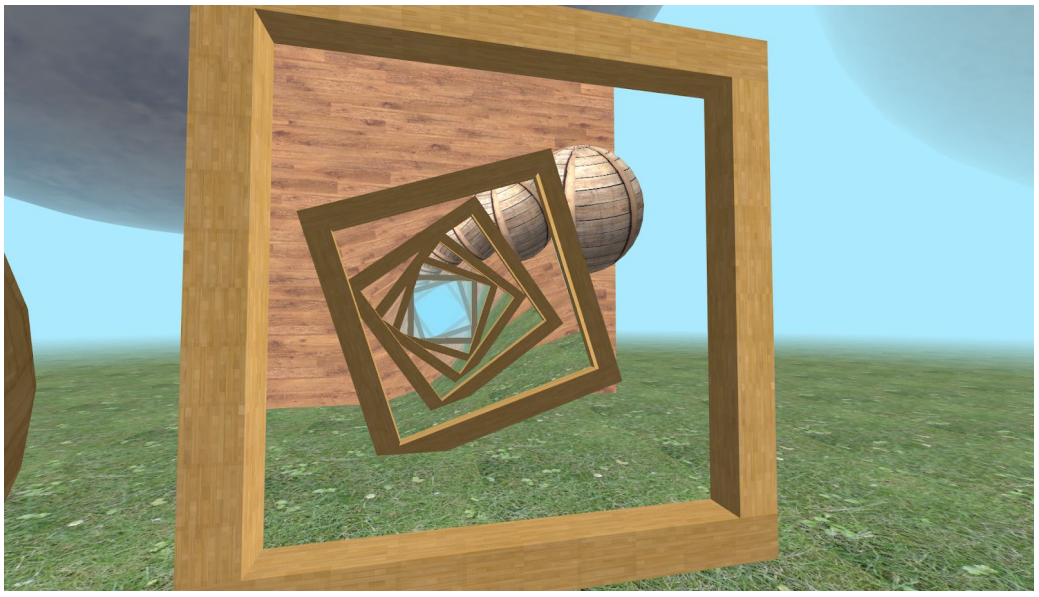
Non-Euclidean Geometry

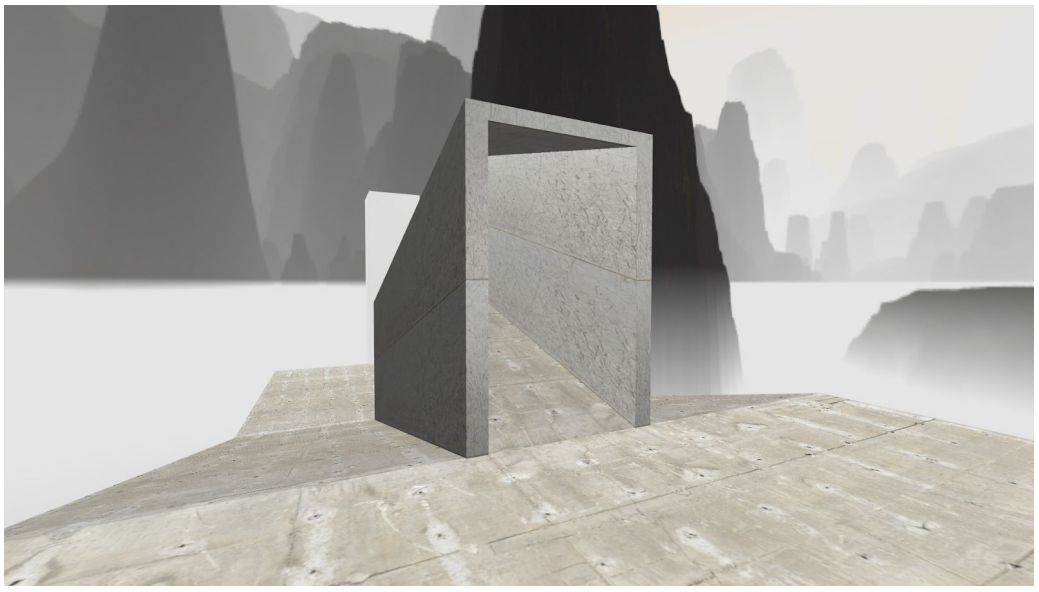
TSBK07 - Vt 2019

Måns Gezelius mange694
Axel Nordanskog axeno840

Source:
https://github.com/Golen87/TSBK07

Demo:
https://golen87.github.io/TSBK07/





1. Introduction

A non-euclidian world is one where the laws of space and time are broken. Examples of such worlds are in the works of Escher, drawings of impossible figures and geometry, or the game Portal where the player connects parts of the world with a portal gun.

Our goal for this project was to write a rendering engine for non-euclidian worlds with the use of framebuffer objects and portal surfaces, seamlessly connecting parts of the world in creative ways.

The project should be web based, using WebGL and Javascript, and should include models, textures and lighting, collision detection and portals.

Optional requirements included the use of portal recursion, frustum culling, occlusion culling, skybox, random generation of mazes, multiple light sources (through portals), puzzle-like gameplay, and sound effects.

2. Background information

What exactly do we mean with a portal? A portal is a surface that interlinks two distant locations, allowing the player to both see through and seamlessly teleport when stepping into it. Portals require us to render the same scene from a different perspective and apply the result to the portal surface texture. The second requirement is teleportation, which is done by multiplying the view matrix with the portal model matrices.

Rendering a scene onto a portal surface is achieved by using framebuffer objects.

3. Implementation

The program was implemented for the web using WebGL and JavaScript. The OBJ file loader *webgl-obj-loader* was used to load models and the matrix and vector library *glMatrix* was used for matrix operations. Collision with walls and objects was handled by the physics engine *Cannon.js*, hereby denoted by “Cannon”.

The program consists of various classes and managers to structure the project into smaller components, such as Preload, Texture, Shader, Physics and Input helpers. Rendering is managed by Scene and Fbo, which continuously render the current scene utilizing objects such as Model, Portal, Camera and PlayerCamera that manage transformations and properties. Each of the 7 example scenes are stored in their own scripts and contain a scene generating function that utilizes helper functions in Scene, such as adding a model with physics.

Framebuffer Objects

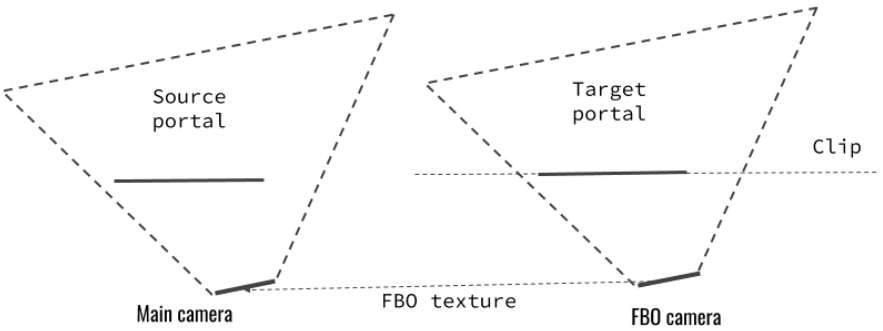


Figure 1: Spawning of a child camera, relative to a (source) portal's target portal, which renderers to a texture using an FBO.

When a portal is in view (see Optimizations) of a camera, a child camera is spawned and transformed using a precomputer portal-to-portal warp matrix to the target portal. See figure 1. This camera will then render the scene to a texture using an FBO. This texture will then be used to render the portal surface in the original camera view.

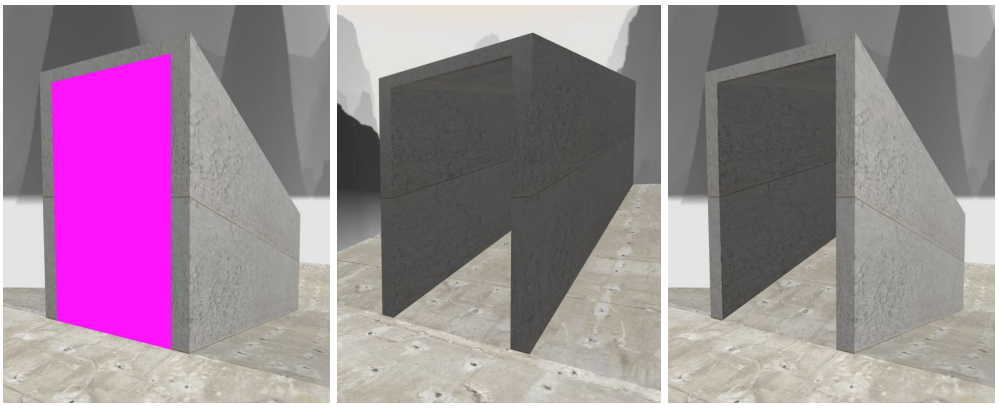


Figure 2: *Left:* The view of a portal. *Center:* The corresponding view, relative to the target portal. *Right:* The target view rendered onto the original portal's surface.

This texture will fit the whole screen and will therefore not be drawn in its entirety onto the portal surface. Instead, the portal surface' vertices' on-screen positions will be used as their own UV-textures to mask the relevant subsection of the texture. This is illustrated in figure 2.

The frustum of the child camera is clipped to position its new plane along its portal's surface. This way, no objects in behind the target portal with the drawn onto the original portal's surface.

This process is repeated recursively for any visible portals from the target portals' perspectives until a specified maximum recursion depth has been reached.

Optimizations

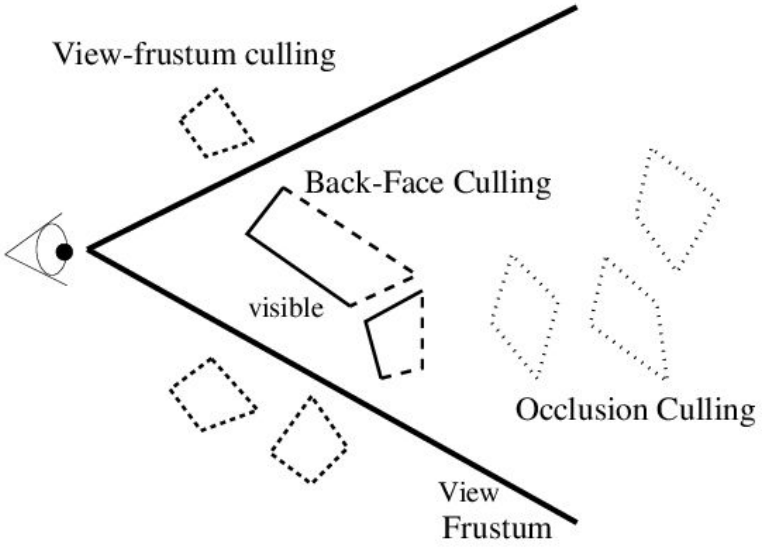


Figure 3: Illustration of different kinds of culling.

Due to portals rendering an entire scene and recursive portals leading to an exponential growth of draw calls, optimizing what should be drawn or not is a must. The focus on these optimizations are about different kinds of culling, see figure 3.

View-frustum culling allows us to remove objects and portals that aren't within the view frustum. This was achieved by calculating the 6 planes that make up the frustum and remove any objects which are outside of one or more planes. Each model has a bounding sphere radius which is used in the plane distance calculations.

Back-face culling allows us to remove any surfaces that are facing away from the viewer. While this feature already exists in OpenGL, we are required to do it manually for portal surfaces to check whether any FBO rendering should be performed. The culling technique simply uses the dot product to the portal surface normal and the viewer direction.

Occlusion culling allows us to remove any objects that are blocked from view, either by an obstacle or a portal surface. Here we render the scene like normal but allow portals to set up a query to check whether any pixels of its surface is drawn or not.

Portals are sorted by distance from the camera and drawn from front to back. This combined with occlusion culling speed up things up and prevents portals hidden behind other portals to be drawn.

Teleportation

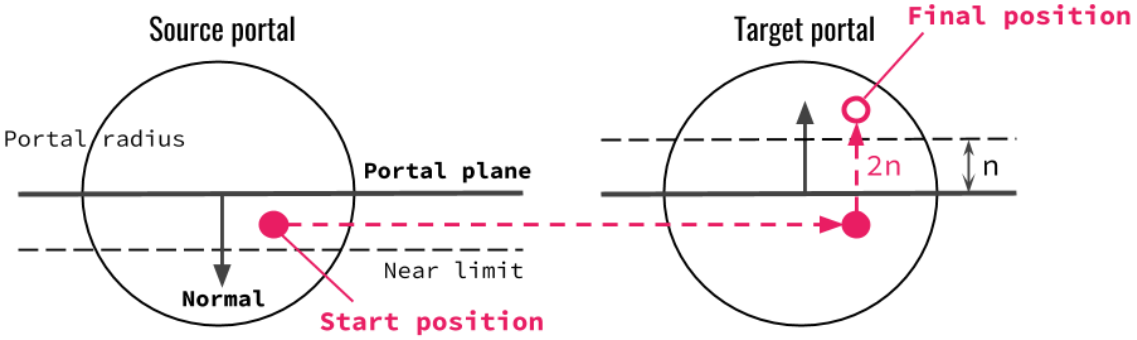


Figure 4: Teleportation of the player from a (source) portal to its target portal.

Teleportation is triggered by the player entering a portal. This is detected by checking if the player has passed a portals *near limit plane* while being within the XZ-plane radius of the portal. The near limit plane is a plane parallel to the portals surface, with a distance to it equal to the projection frustum new plane offset. This results in a portal surface never being clipped by the frustum while outside its near limit planes. and teleporting the player when passing this guarantees that they will always be outside it.

Teleportation is performed by applying a precalculated portal-to-portal warp matrix to the player's position and rotation respectively to have the player moved to the relative position behind the target portal. The player is then offset past the target portal's surface and near limit plane along its normal. This process is illustrated in figure 4.

Since this whole operation is performed in full before rendering the scene again, the player is never visibly inside the near limit plane of either the source portal or the target portal, this totally preventing clipping of portal surfaces.

4. Interesting problems

One of our first problems was figuring out how to apply an FBO to a portal texture correctly and not have it seem like a TV monitor showing a scene. The solution was so render the full screen and use the uv coordinates in the fragment shader, which feels more like masking.

Drawing portals very close to the camera caused some issues, as the near clipping plane cut into the surface and drew both worlds at once. This was later solved with a smaller near limit and teleportation preventing the player from ever moving too close to a portal.

Query results have a delayed result of approximately 1-3 frames, depending on browser and computer speed, which caused issues. As of such, queries had to default to an occluded status to prevent unnecessary draw calls, which causes a slight delay when new portals are first seen.

Occlusion culling during portal recursion caused a lot of problems. In order to solve it, each portal object has a list of active queries for each combination of parents that request it to be drawn. However, this causes portals to glitch upon teleportation due to the occlusion queries giving a different result for a previously blocked portal. This issue was solved by fetching the occluded state of each portal for the previous frame, regardless of depth, and assume those that were visible before teleportation are visible after. This causes a noticeable delay upon teleportation in scenes with 20+ portals, but perfectly removes any ugly frame glitches.

Due to a problem with gravity causing the player’s position to jitter when standing on the ground, walking on the ground was instead solved by raycasting towards the ground and have the player snap to and walk along the hit surface.

Cannon utilizes its own set of vector and matrix structures separate from the glMatrix library, which caused some conversion issues. Cannon was used for operations on the player’s position and orientation while all other calculations were done using glMatrix.

5. Conclusions

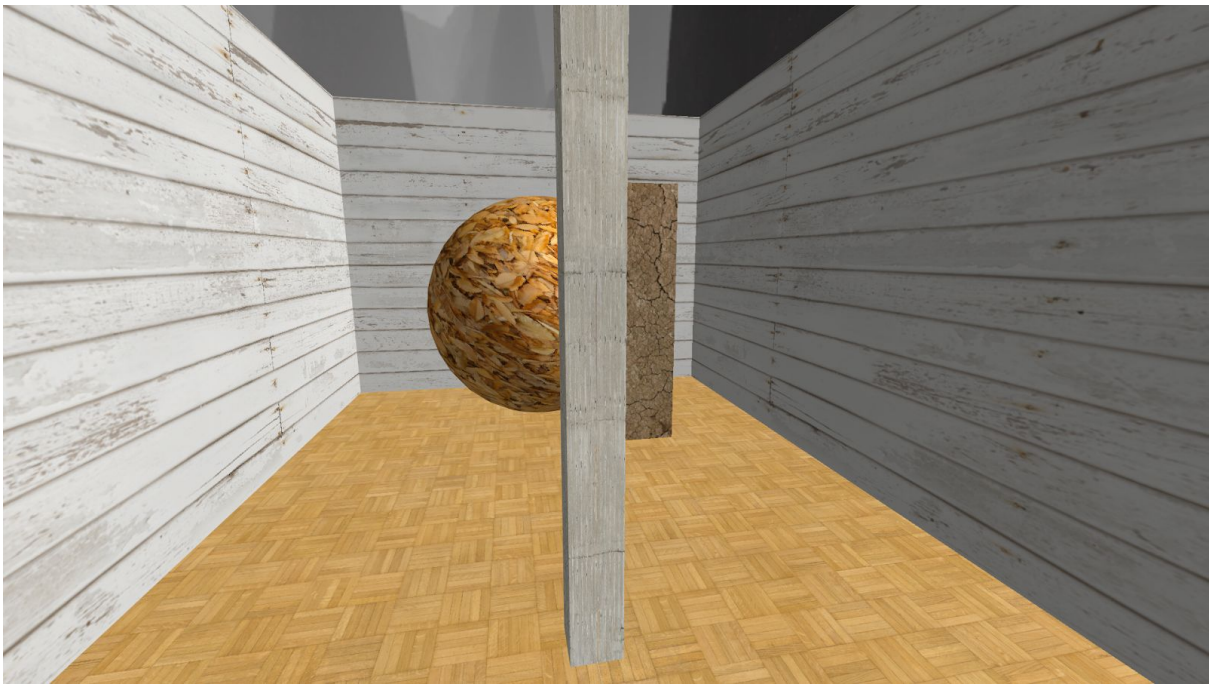


Figure 5: Final rendering of scene with multiple rooms, containing different objects, that are moved between by circling around a pillar. The right half of the screen is another room rendered through a portal. Along the center of the pillar, a vertical slight gap can be seen which is located along the a portal edge.

Renditions inside portals are drawn one pixel above where they should be, there are very narrow gaps along portal edges and portal contents are rendered somewhat sharper compared to the objects rendered directly onto the screen (from the main camera).

The end result is however convincing overall. Without knowing what to look for, and actively doing so, it is easy to miss what parts of the view is made up of portals. Judge for yourself by studying figure 5. The small step through portals to avoid being inside their near limit planes is also not noticable.

Potential improvements

A potential further optimization would be to limit the frustum of child cameras so that its wides touches the portal frame. This would result in only objects and portals actually visible through the portal from the original camera passing frustum culling checks.

Also, sorting of portals is only done for the main camera which means that portals can be in reverse order from child cameras' perspectives which in turn results in reduced performance. Sorting portals for each camera might therefore improve performance.

It could be interesting to add support for moving objects through portals. Portals themselves could also be moved if they their warp matrixes are recalculated. Currently these are only calculated when portals are initially set up. If support for changing the down direction for ground raycasts is added, the player can also be made to walk along the ceiling or walls after passing through portals. Similarly, the size of the player can be scaled if the offset from ground to camera can be changed.

Support for point light and shadows would also be interesting but these would need to shine/be cast through portals.