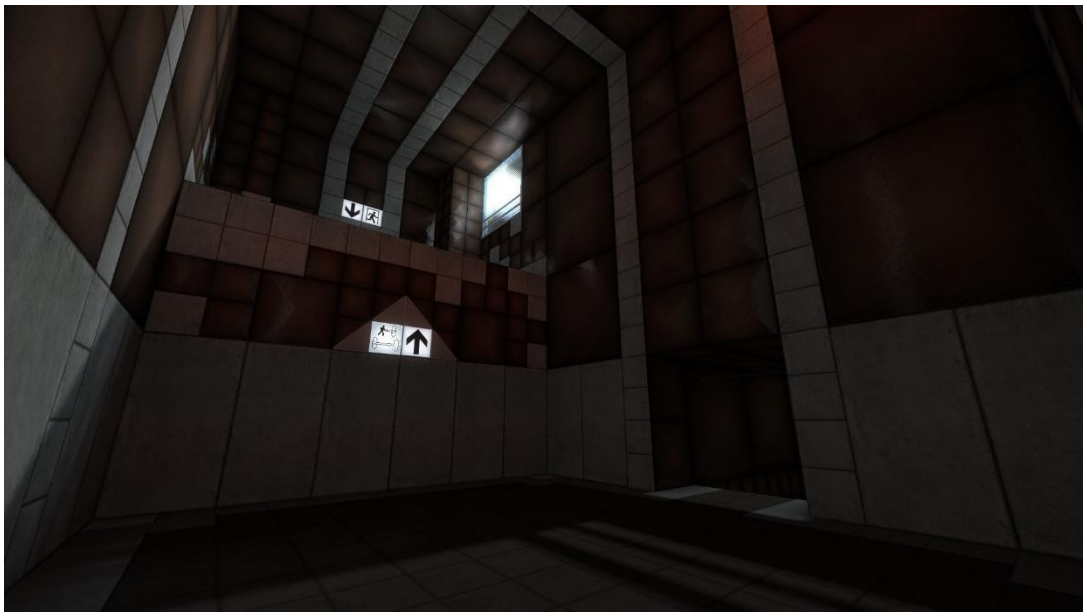
Implementing Advanced Rendering Techniques in Valve's Source Engine

Implementing enhanced specular reflections for ***Portal: Outside Influence***

Written by Caylee Morris ([REDACTED])



Screenshot from Portal: Outside Influence, showing parallax corrected reflections (Elite Treacle Ltd., 2018)

Abstract

Videogames in development will frequently have their technology locked in ahead of time, be it due to licensing constraints, cost concerns or other forces. Regardless of this, it's in the best interests of all involved to produce as polished a product as possible, and with the ever-moving expectations of customers, this polish will largely be seen in the visuals of the title. The goal of the research and implementation conducted here is to improve the graphical fidelity of the *Source Engine* by *Valve Software* for use in the title *Portal: Outside Influence* by *Elite Treacle Ltd.,* focusing on improving the rendering of image-based specular reflections throughout the game world without requiring major modification of existing game assets or content.

Table of Contents

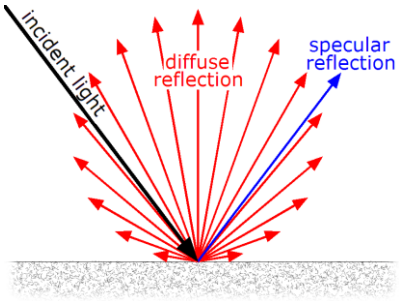
Introduction

When starting development on any software project, it is important to lock-in a technology base or set of technologies during development and stick with it throughout. This holds true for videogames as with any other software, with the choice of game engine (the development platform on which the entire project is created) being locked in relatively early on in most projects. Moving engine is a time consuming and risky task, often requiring whole sections of work be entirely thrown away. Usually a game engine is chosen for its feature set, hardware platform compatibility, team experience, or cost. Sometimes, an engine is decided based on licensing terms. In the case of *Portal: Outside Influence*, (Elite Treacle Ltd., 2018) the engine choice was straightforward: being based on a Valve Corporation intellectual property (that being *Portal,* their title released as part of *The Orange Box*), (Valve Corporation, 2007) and being partially based on the mechanics of that series, it made sense to utilise the same engine. This made for a simpler licensing arrangement, as well as provided a mature platform to work with that already had core game functionality built-in. However, *Portal 2,* the engine branch upon which Outside Influence is based, was released in 2011. 7 years and an entire game console generation (and a half) later, the rendering capabilities have begun to look dated in some areas. With the release of the current generation of consoles starting in 2013, the standard for rendering in games has risen, with users now expecting environments and objects to be lit in a more accurate manner than is offered by Source in its standard form. The goal of this project was to bring the scene rendering quality in Portal: Outside Influence up-to par with similarly priced games on the market today, without rewriting the majority of the render pipeline and renderer, and without requiring large-scale reauthoring of existing game assets.

Half-Life 2 (top-left), Portal 2 (top-right), Counter-Strike: Global Offensive (bottom-left), Left 4 Dead 2 (bottom-right) (Valve Corporation, 2004) (Valve Corporation, 2011) (Valve Corporation, 2012) (Valve Corporation, 2009)

Background

Diffuse and Specular Reflections



Specular and diffuse reflections. (GianniG46, 2010)

In the real world, surfaces reflect light in straight lines, with that reflection being dependent on the nature of the surface. Some objects may exhibit clear reflections (known as *specular*), while others may exhibit a scattered reflection with no clear image (known as *diffuse*). Rendering a 3D scene generally involves simulating both forms of reflection; with an object's *albedo colour map* texture and a diffuse lighting calculation (such as *lambert*, *half-lambert* or *phong*) to simulate the diffuse reflection, with a specular reflection calculated separately. The specular reflection is where interest lies for this implementation, as Source already features a number of diffuse models that are serviceable for the needs of the game. Traditionally, specular reflections are image-based while diffuse reflections are based on a variety of factors, sometimes utilising a pre-filtered image-based solution, or sometimes just adjusting the light values of the vertices of the model being rendered.

All 3D rendering is based on the premise of the *rendering equation*, attempting to emulate the way light rays interact with surfaces in as computationally cheap a manner as possible; running a real-time application using the actual equation would simply not be practical due to the speed at which such applications need to run. (Immel, et al., 1986) (Kajiya, 1986)

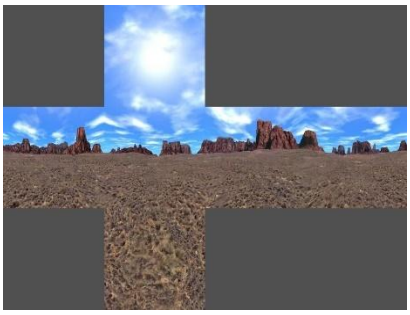
$$L(\mathbf{x},\vec{\omega}_o) = L_e(\mathbf{x},\vec{\omega}_o) + \int_S f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o) L(\mathbf{x}',\vec{\omega}_i) G(\mathbf{x},\mathbf{x}') V(\mathbf{x},\mathbf{x}') d\omega_i$$

,where $L(\mathbf{x},\vec{\omega}_o)$ = the intensity reflected from position $\mathbf{x}$ in direction ω_o
$L_e(\mathbf{x},\vec{\omega}_o)$ = the light emitted from $\mathbf{x}$ by this object itself
$f_r(\mathbf{x},\vec{\omega}_i \rightarrow \vec{\omega}_o)$ = the BRDF of the surface at point $\mathbf{x}$, transforming incoming light ω_i to reflected light ω_o
$L(\mathbf{x}',\vec{\omega}_i)$ = light from $\mathbf{x}'$ on another object arriving along ω_i
$G(\mathbf{x},\mathbf{x}')$ = the geometric relationship between $\mathbf{x}$ and $\mathbf{x}'$
$V(\mathbf{x},\mathbf{x}')$ = a visibility test, returns 1 if $\mathbf{x}$ can see $\mathbf{x}'$, 0 otherwise

The rendering equation and its components
(Immel, et al., 1986) (Kajiya, 1986) (Yeunglm, 2011)

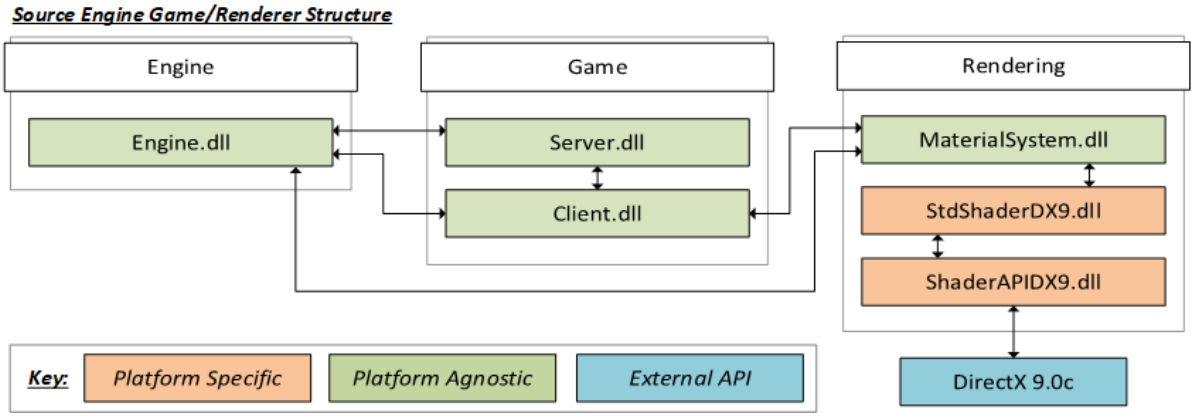
The Source Engine

Valve Corporation's Source Engine was first released in 2004 with its flagship title, *Half-Life 2*. (Valve Corporation, 2004) Based on their previous *GoldSrc* engine (itself based on *idTech 2* used in *Quake),* (id Software, 1996) it boasted an expansive feature set for the time, including support pixel and vertex shaders, high quality lightmaps, render-textures, and versatile materials with support for bump-mapping, detail maps and image-based reflection mapping via offline-compiled cubemap textures. Cubemap images are also used for 2D skyboxes, just as in their earlier title *Half-Life.* (Valve Corporation, 1998) *Half-Life 2: The Lost Coast* brought high dynamic range *(HDR)* lighting, (Valve Corporation, 2005) and *Half-Life 2: Episode One* brought improved radiosity calculation during map compiles and further HDR improvements. (Valve Corporation, 2006) With *The Orange Box*, the



Cubemap image used to render a sky in Half-Life (Valve Corporation, 1998)

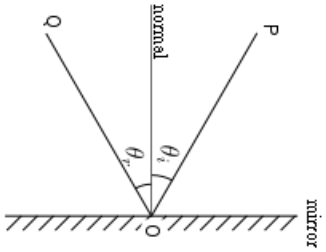
engine got an upgrade to support an even wider set of rendering features, such as a phong shading model for game entities, dynamic shadowing flashlights via a projected texture, and higher quality lightmaps. (Valve Corporation, 2007) Future titles would add more features, including flowmaps on water surfaces in *Left 4 Dead 2* (Valve Corporation, 2009) and improved dynamic shadow filtering in *Alien Swarm.* (Valve Corporation, 2010) However, the methods by which image-based reflection mapping (and indirect lighting as a whole) are handled are essentially unchanged since 2006 (this remains true of even Valve's most recently updated titles on this engine, such as *Counter-Strike: Global Offensive)*. (Valve Corporation, 2012) Source makes use of the *DirectX 9* graphics API (with limited support for DirectX 7 and 8) and an *OpenGL* translation layer called *'ToGL'* on non-Microsoft platforms. (Valve Corporation, 2014) This is a limiting factor when it comes to deciding how to improve the rendering of the engine as many modern features, such as *compute shaders* and native *raytracing* is not supported. Rewriting the renderer to support modern versions of DirectX or another API such as V*ulkan* is out of scope for this project; according to an anecdote from a Valve employee, the only successful externally-authored rewrite of Valve's render pipeline was by *Respawn Entertainment,* which saw a move to the DirectX 11 API implemented by a team of 10 over the span of 6 months for use in *Titanfall*. (Respawn Entertainment, 2014)



Engine.dll is shared for multiple games, Server and Client are game-specific, MaterialSystem deals with textures and material-based rendering, ShaderAPIDX9 interfaces between the render system and DirectX 9 and StdShaderDX9 contains the C++ functionality for setting up specific shaders (in fact, there can be additional shader DLLs should a game require them)

Simple Image-based Specular Reflection Mapping

The most basic method of producing a specular reflection effect is to make use of a *cubic environment map texture (cubemap)*. This is a set of 6 images with a 90-degree field of view and square aspect-ratio, taken at the same point in space but at 90-degree offsets from one another. This set of images forms a complete, rotatable image of the surroundings of the sample point. This image can be used to perform various effects. Here, it uses the reflection vector calculated from the camera view position and world-space normal of the current polygon (perturbed by a *normal map texture,* should the polygon have one



(Arvelius, 2005)

applied). The cubemap is sampled based on this calculated vector, and this is then blended with the rest of the material (diffuse lighting, detail textures, etc.), either with a basic additive operation or by using a more advanced technique such as applying a *Fresnel* function. The equation for calculating the reflected angle used in all Source shaders is as below:

$$2 \times \frac{normal \cdot eyeVector}{normal \cdot normal} \times normal - eyeVector$$

Cubemaps are not the only available form of environment map, (spherical and HEALPix maps are some alternatives) but it is by far the most commonly seen in games due to having the best trade-off between required texture inputs and distortion, as well as the ease at which the reflection vector of a surface is translated into a cubemap sample coordinate.

Specular reflections on models in Half-Life 2 (Valve Corporation, 2004)

This approach is quick to render, light on memory, easily implemented in every modern graphics API, and can be baked ahead of time by pre-compiling the cubemaps for the world at build time and saving them for later use. These pre-calculated cubemap images can be rendered at a dramatically higher quality than the system the game is intended to be running on or have effects applied during the compile process that would be too expensive to perform at runtime. However, this approach has its drawbacks, the most important of which being the spatial inaccuracy of the reflection. A cubemap is only sampled from a single point, with the reflection becoming less and less accurate the further the reflective object moves from that point, demonstrating *'infinite parallax'* as no adjustments are made based on the relative positions of the object and cubemap sample. While outdoor environments can often make-do with this limitation (game skyboxes are frequently just cubemaps that render in the background of a scene as it is), certain surfaces would not look correct, such as those near bright light sources or large, reflective planes. Other issues with prebuilt cubemaps are that convex objects will not self-reflect, dynamic entities are not included in the reflected image, and the reflection is not updated to respect a change to the environment itself (such as a light turning on or off). These are usually less noticeable than the parallax inaccuracies, but should still be considered when looking at alternative solutions.

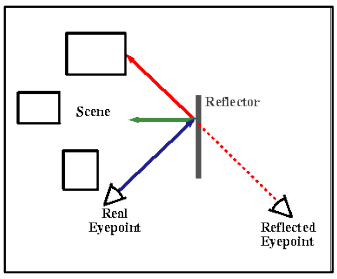
Alternate Specular Reflection Mapping Techniques

While Source largely relies on the above simple cubemap technique for its specular reflections, there are alternative that should be considered before attempting to iterate on the pre-existing approach.

Planar Reflections

Planar reflections on water in Half-Life 2 (Valve Corporation, 2004)

A simple technique for creating a reflection in a scene is to render the scene again for every required reflection. This is a costly method, requiring a substantial portion of the scene to be rendered from a perspective relative to the reflection surface and main camera, the clipping plane of the reflected camera set to that of the reflection surface. That output is sent to a texture for use in the final scene render. It is most commonly used on large surfaces that require accurate spatial reflections and world-entities to be truly visible in the image, such as bodies of water or a mirrored floor. Due to the nature of the rendered image, the reflected scene must be drawn for every unique surface, which can quickly become slow and impractical to render. As such, engines tend to have strict limitations on surfaces using this technique; Source only allows one dynamic planar visible per-scene, requiring reflections of varying heights or angles to be completely separated from each other. It is not a technique viable for use on many objects in a scene, nor is it realistically possible to render reflected images for every vertex on a complex model fast enough to perform the operation every frame.



(McReynolds, et al., 1997)

In order to capture the reflection image, the plane of reflection is taken and used to mirror the current camera's position. The near-distance clipping plane is set to the plane of the surface, preventing anything underneath from being reflected too. The scene is then rendered to a texture, which is then applied to the desired reflective surface. (Epic Games, 2018)

Screen-space Reflections

A screenspace reflection depth error in ***Infestation: Survivor Stories*** *(formerly* ***The War Z****)* (Hammerpoint Interactive, 2012)

Screen-space Reflection is a technique that takes advantage of a *'g-buffer',* which stores material and geometry information about a scene within screen-space, as well as the rendered scene image, which at this point lacks any specular reflections. Reflections are not rendered with the scene, but instead deferred and rendered on top of the scene as a post-processing effect. It is performed by using the surface normals and material information stored within the g-buffer to calculate a reflection vector and perform screen-space ray-traces against the depth information of the scene, and use that position to sample the rendered scene. This results in a reasonably fast and dynamic reflection that can be drawn on all objects in the game and does not require pre-calculation. However, this solution has numerous drawbacks; for one, it requires a g-buffer to function. While g-buffers are a required part

of a deferred renderer, many forward renderers do not write to such a buffer. Another issue stems from building the reflection from the scene image, preventing anything that is off-screen or otherwise obscured by other geometry from being rendered in a reflection. Poorly authored screen-space reflection shaders can lead to jarring, inaccurate and humorous results, such as those seen in *Infestation: Survivor Stories* (formerly *The War Z,* pictured). (Hammerpoint Interactive, 2012) Effectively, this means that screen-space reflections need to be blended with other reflection techniques in order to be truly viable.

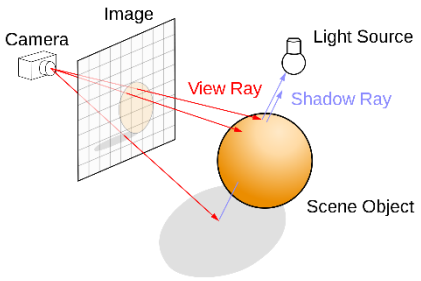
***Grand Theft Auto V**'s G-buffer components* (Courrèges, 2015)

For the purposes of implementation in Source, this wasn't an easily authored solution and would require heavy modification to parts of the render pipeline and shader set, including re-enabling a g-buffer previously only used for some optimization steps on console hardware, and removal of current specular reflection methods from the standard shader set. Alongside this, various functions commonly used by screen-space reflection implementations are not available in the DirectX 9 graphics API, and would need to be re-implemented using available functionality.

Ray-Traced Reflections

***Star Wars: Reflections** raytracing demo in Unreal Engine 4* (Epic Games; Nvidia; ILMxLAB, 2018)

While screen-space reflections are based on simulating ray-traces within an existing scene image, *ray-traced reflections* go a step beyond this by performing ray-traces against the full geometry of the scene. In order to accurately simulate a true simulation of light in real-time, hundreds of rays will need to be calculated per-pixel. As a result, ray-traced rendering has typically been reserved for offline render calculations using powerful processors or render-farms, such as the lightmap compilation step in the Source level compile process, or rendering CG theatrical productions. While a *Monte Carlo* method of sampling, denoising step and some other tricks can be used to cut down the samples needed to a far lower number and produce an acceptable result, the operation of ray-tracing against high-detail level geometry is still a slow operation for current graphics hardware. NVIDIA, AMD, Khronos Group and Microsoft are all working on ray-tracing implementations with hardware acceleration, however these all require the use of modern graphics APIs and the very newest hardware, limiting the users that could make use of such effects. As a result, it has never been the only available implementation for specular reflections in a shipped game, and is usually an afterthought to demonstrate new technology added post-launch. *Battlefield V* is one such game, featuring use of *Nvidia RTX* accelerated raytracing, but falling back to more traditional methods if the user is not running a supported GPU. (Nvidia Corporation, 2018) (EA DICE; Criterion Software, 2018)



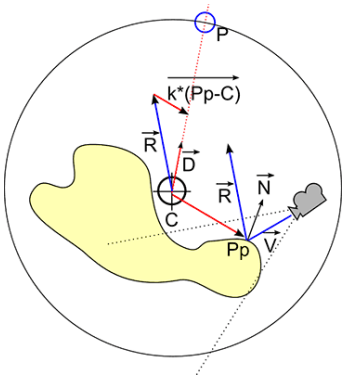
A simple ray-trace against a sphere with a single light source (Henrik~commonswiki, 2008)

Enhancing Cubemaps

While the previously discussed methods have their merits, they aren't viable for the bulk of reflections that would be seen in Portal: Outside Influence, be it due to a graphics API limitation or lack of versatility. Instead, methods of improving the existing image-based methods should be considered.

Spherical Parallax Correction

An early attempt to fix the infinite parallax issue faced with standard cubemaps made use of a radius around the point of the cubemap sample and adjusted relative to the position of the object. This concept has been published in the context of real-time several times, once in 2002 using *shader assembly language*, (Brennan, 2002) and again later in *HLSL (DirectX High Level Shading Language)* (Bjorke, 2004), seemingly based on a section of a book on offline-rendering called *Advanced RenderMan*. (Apodaca & Gritz, 1999) This technique solves the parallax issue by using a sphere as *proxy geometry* on which to adjust the reflection ray against. Due to the nature of the shape, only perfectly matches spherical environments. As the majority of game environments are not spherical (especially in games of the time, which were largely boxy affairs often utilizing BSP geometry constructed from flat-sided primitives), this would not be an adequate solution for the parallax problem. It would only really be suitable for a game featuring a more natural-looking outdoor environment or with very careful placement of cubemap sample points.



(Lagarde & Zanuttini, 2012)

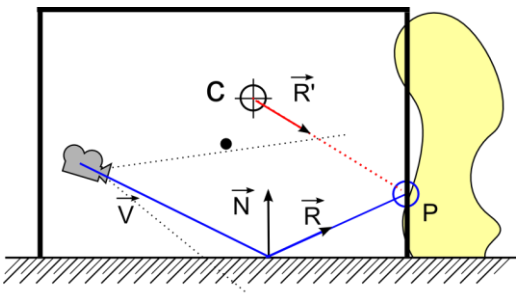
The shader code for applying this method of correction is brief, as seen below:

```
ReflDirectionWS = EnvMapOffset * (PositionWS - CubemapPositionWS) + ReflDirectionWS;
```

(Apodaca & Gritz, 1999) (Brennan, 2002) (Bjorke, 2004) (Lagarde & Zanuttini, 2012)

Bounding Box Parallax Correction

To properly correct for the infinite parallax in cubemap reflections, the shader must be aware of both the image sample point and the local geometry in the scene. However, this is a non-trivial task due to the amount of data and compute time required to test against every polygon in a scene. As mentioned above, earlier attempts were made to simplify this by use of a sphere radius, but eventually Bartosz Czuba experimented with using an axis-aligned bounding box to act as proxy geometry; a version of the environment that is roughly accurate to the intended geometry, but simplified so that math can be performed upon it in the shader. In this case, they made use of an Axis-aligned Bounding Box (*AABB*) and performed a simulated ray-trace against it to correct the parallax. (Czuba, 2010)



(Lagarde & Zanuttini, 2012)

Below is a shader sample for applying this form of correction, requiring the cubemap position, box position and box extents:

```
    float3 DirectionWS = PositionWS - CameraWS;
    float3 ReflDirectionWS = reflect(DirectionWS, NormalWS);

    // Following is the parallax-correction code
    // Find the ray intersection with box plane
    float3 FirstPlaneIntersect = (BoxMax - PositionWS) / ReflDirectionWS;
    float3 SecondPlaneIntersect = (BoxMin - PositionWS) / ReflDirectionWS;
    // Get the furthest of these intersections along the ray
    // (Ok because x/0 give +inf and -x/0 give -inf )
    float3 FurthestPlane = max(FirstPlaneIntersect, SecondPlaneIntersect);
    // Find the closest far intersection
    float Distance = min(min(FurthestPlane.x, FurthestPlane.y), FurthestPlane.z);

    // Get the intersection position
    float3 IntersectPositionWS = PositionWS + ReflDirectionWS * Distance;
    // Get corrected reflection
    ReflDirectionWS = IntersectPositionWS - CubemapPositionWS;
    // End parallax-correction code

    return texCUBE(envMap, ReflDirectionWS);
```

(Czuba, 2010) (Lagarde & Zanuttini, 2012)

This technique was severely limited however, locking level geometry to the world axis. In 2012 this technique was improved by Sébastien Lagarde and Antoine Zanuttini for use in the game *Remember Me.* (Lagarde & Zanuttini, 2012) They expanded the technique to use an Object-aligned Bounding Box (*OBB*), which frees level designers from grid alignment and makes the correction far more useful for real game environments. While this does require slightly more complex maths and additional data to be passed through to the shader, the change is basically negligible to modern graphics hardware. In order to achieve this, the pair perform their calculations within the local space of the box itself, requiring simply the use of an inverse transformation matrix generated from the OBB and transforming the world-space reflection ray into box-space, performing basic intersection tests within the unitary bounds of the box (-1.0 to 1.0 along each axis). The shader code for this method of correction is rather similar to the AABB method, but with adjustments for transforming between world and box-space, as seen here:

```
    float3 DirectionWS = normalize(PositionWS - CameraWS);
    float3 ReflDirectionWS = reflect(DirectionWS, NormalWS);

    // Intersection with OBB convertto unit box space
    // Transform in local unit parallax cube space (scaled and rotated)
    float3 RayLS = MulMatrix( float(3x3)WorldToLocal, ReflDirectionWS);
    float3 PositionLS = MulMatrix( WorldToLocal, PositionWS);

    float3 Unitary = float3(1.0f, 1.0f, 1.0f);
    float3 FirstPlaneIntersect  = (Unitary - PositionLS) / RayLS;
    float3 SecondPlaneIntersect = (-Unitary - PositionLS) / RayLS;
    float3 FurthestPlane = max(FirstPlaneIntersect, SecondPlaneIntersect);
    float Distance = min(FurthestPlane.x, min(FurthestPlane.y, FurthestPlane.z));

    // Use Distance in WS directly to recover intersection
    float3 IntersectPositionWS = PositionWS + ReflDirectionWS * Distance;
    float3 ReflDirectionWS = IntersectPositionWS - CubemapPositionWS;

    return texCUBE(envMap, ReflDirectionWS);
```

(Lagarde & Zanuttini, 2012)

While the correction is still limited to a box-shaped proxy, the ability to rotate the box offers just enough accuracy for most many game environments, especially those constructed in BSP-driven engines such as Source, which features environments largely constructed from flat-sided shapes.

Parallax Correction via Custom Proxy Geometry

An expansion on the methods previously discussed is performing parallax correction via the use of custom proxy geometry. This has been implemented in far fewer titles than the OBB method (which sees wide support in modern game engines) due to requiring the use of modern graphics APIs to help pass additional meshes of arbitrary size to the pixel shader in order to perform intersection tests against it. This method is largely the same as the OBB method, testing against an arbitrary number of planes or triangles instead of the 6 planes that construct an OBB. This offers more accuracy in reflections, allowing for non-cubic level geometry to be properly represented, however requires far more information to be passed to the shader, something that is not a native DirectX 9 feature. This is technically feasible in DirectX 10 and above, allowing for data to be passed in a *constant buffer*. This has been implemented in other titles, but would run slower than the OBB method due to the additional complexity of testing against a more complex shape, and may simply not be possible in DirectX 9 at all due to the difficulty of passing in the required mesh data, and the restrictive nature of the shader compiler requiring code loops to have a fixed number of iterations at build time. This has been verified by feedback received from Sébastien Lagarde (the original author of the OBB and mesh-based parallax correction technique) and Krzysztof Narkowicz (graphics programmer at Epic Games, previously lead engine programmer at Flying Wild Hog on *Shadow Warrior 2*). (Lagarde, 2018) (Narkowicz, 2018)

Ray-plane intersection tests are relatively simple computationally. The dot-product of the plane normal and ray normal is calculated, with that then compared against an *epsilon value* (a very small value that is so close to zero it may as well be considered zero). If the dot-product is above this epsilon value, then the ray will intersect at some point, and the intersection point can be recovered using the ray origin, ray normal, plane position, plane normal and the earlier computed dot-product value, as seen below:

```
    bool IntersectRayPlane(float3 rayOrigin, float3 rayDirection, float3 planePosition, float3
planeNormal, out float3 intersectionPoint)
    {
        float rDotn = dot(rayDirection, planeNormal);
        if (rDotn < 0.0000001f ) {
            return false; // no intersection occurred; facing away or parallel
        } else {
            float s = dot(planeNormal, (planePosition - rayOrigin)) / rDotn;
            intersectionPoint = rayOrigin + s * rayDirection;
            return true;
        }
    }
```

(tonfilm, 2013)

Implementation

Overview

In order to decide which implementation should be approached for use in Portal: Outside Influence, the capabilities of both the Source Engine, the requirements of any given method, and the overall quality of the results generated by the method.

Solution Requirements

Using the *MoSCoW method*, (Madsen, 2017) a set of guidelines for the implementation can be laid out.

MUST:

- *…improve the visual quality and accuracy of specular reflections*
- *…be supported by both static world geometry (BSP brushes) and by dynamic game entities*
- *…be implemented in all standard world and object shaders (LightmappedGeneric, VertexLitGeneric, Refract, Phong)*
- *…require as little change to existing assets and render pipeline as possible (limiting to DirectX 9 as graphics API)*
- *…work on the lead platform (Microsoft Windows)*

SHOULD:

- *…make only a small impact on overall framerate (game should maintain 60fps on low-spec hardware; 6-year-old laptop with low-mid range GPU)*
- *…integrate with the Hammer World Editor to allow easy addition to game worlds*
- *…not require level designers or artists to use external tools beyond those already utilized in the pipeline as-is*

COULD:

- *…work on other target platforms (MacOS, Linux)*
- *…be reusable for other purposes in the game*
- *…not inflate size of game data or level files far beyond the current cubemap solution*

WON'T:

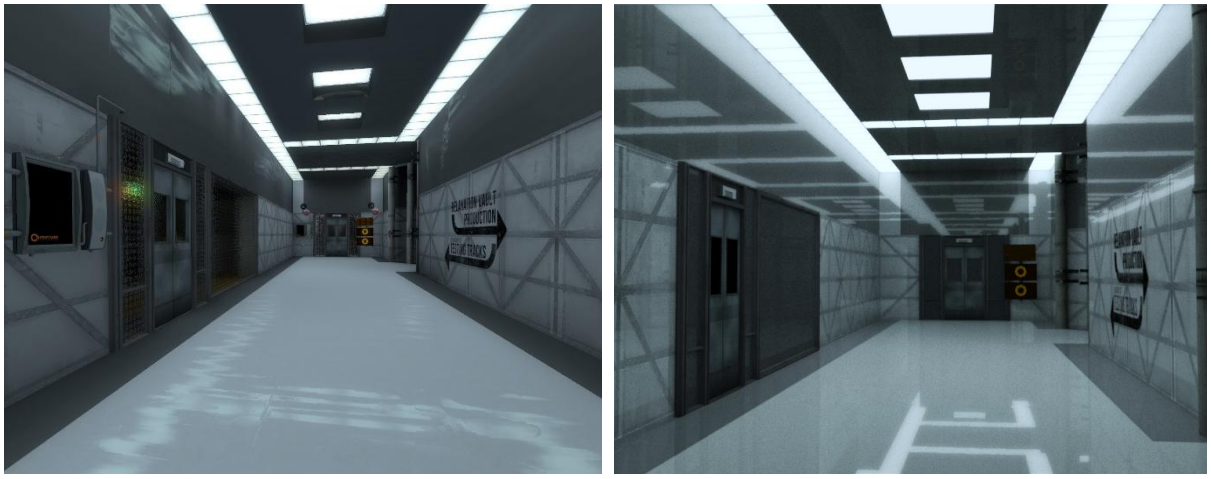
- *…drastically modify how reflection sampling is handled in the Hammer World Editor*
- *…replace the Source Engine renderer, upgrade the graphics API, or dramatically alter the render pipeline*

Upon comparing these requirements, various methods can be immediately discounted. Ray-tracing cannot realistically be considered; not only does Source's chosen graphics API, DirectX 9, not have the required functionality, but most users will not have the hardware required to run the game at full speed should this be attempted. Planar reflections are already implemented in Source for water and mirror surfaces, however are strictly limited to flat, static world brushes and cannot be considered for rendering of specular reflections on game entities and non-planar world surfaces. Realistically, the best option is to attempt to improve the existing cubemap methods built into the engine via a parallax correction step, as the Source Engine render pipeline already has features for

sampling and packing samples into the world data, baking cubemap reference data on static surfaces, and updating dynamic objects with appropriate local cubemap samples.

After considering the style of environments seen within the game, the object-aligned bounding box correction method was chosen. This allows level design to work away from the cardinal grid, but still fits the relatively boxy environments. Spheres would not be a good fit as the game does not feature spherical environments, and more complex geometry would require passing a large amount of data into the pixel shader, which does not seem technically feasible in DirectX 9, coupled with the serious cost of performing intersection tests against complex meshes for every pixel of specular reflection.

Visual Target



Uncorrected cubemaps in-engine (left), the 3DS Max ray-traced visual target (right)

In order to properly scope the solution, these images will be used. On the left is an image of the stock Source Engine cubemap implementation, with visibly inaccurate reflection parallax. On the right is the visual target, rendered in *3DS Max 2018* (Autodesk, 2018) using the same assets, imported through *Wall Worm Model Tools* (Olson, 2012) and rendered via a slow ray-traced method, representing the ideal appearance of reflections and a goal to aim for with the new implementation. While not a quantitative measure by any means, and the visual target image lacks the same method of distorting the reflection through a normal map, comparison with this visual target is the best way to judging the accuracy of the environment reflection effect to be implemented.

Prior Works

Precedent for the use of this method to improve reflections have been set in the past. The object-aligned bounding box and convex geometry techniques were both developed for Remember Me, making use of them throughout the game on previous-generation console hardware and graphics APIs. Since then, many other titles, including Shadow Warrior 2 made use of the same techniques. Shadow Warrior 2 expanded the method to allow for non-uniform spheroids instead of uniform spheres, and used the corrected cubemap data to improve indirect light simulation in their image-based lighting and specular pipeline, blending with other forms of reflection where needed. (Narkowicz, 2017) Many titles blend a combination of techniques; for instance, *Rainbow Six: Siege* (Ubisoft Montreal, 2015) features a screen-space reflection technique, but falls back to parallax

corrected static cubemaps on non-important world geometry or when the required screen-space data is unavailable to produce the reflection.

The object-aligned bounding box form of parallax correction has also been implemented in many modern engines by their internal development teams, with Unity 3D, (Unity Technologies, 2005) Unreal Engine 4 (Epic Games, 2014) and Source 2 (Valve Corporation, 2014) all seeing fit to add it as a core feature of their specular reflection and image-based lighting solutions. The developers of these engines have not decided to add support for the convex-mesh version of the solution, likely due to the additional overhead of passing the data to the shader and testing against the surfaces.

An implementation of the object-aligned bounding box method has been publicly attempted in the Source Engine once before. Brian Charles released a proof of concept implementation using the publicly available *Source SDK* and demonstrated the capabilities in a short video. Their implementation is not integrated with the built-in shaders, instead opting to demonstrate using an unlit shader, and only supports application on static world geometry as the local cubemaps and their matrices are baked during the map compile process. (Charles, 2013) This provides a useful reference for progressing with an improved implementation, including a method of retrieving the required OBB volume from an easily manipulated Hammer Editor brush shape.

Uncorrected reflections (left) and Brain Charles' parallax correction (right) (Charles, 2013)

Implementing Parallax Correction with Object-Aligned Bounding Boxes

In the Shader

The shader code for parallax correction with object-aligned bounding boxes is relatively short, and allowing for the introduction of a common shader function based of the code sample seen within the original publication by Sébastien Lagarde. By splitting the code into a shared function, parallax correction could be quickly and easily added to a variety of shaders with minimal effort. From the originally published implementation, only minor tweaks were required, including calculating within slightly different bounds (0.0-1.0 along each axis instead of -1.0-1.0) due to a difference in how parallax correction volumes are being calculated.

```
float3 ParallaxCorrectReflectionVector(float3 reflectVec, float3 cubemapWorldPos, float4x4 obbMatrix) {
    // Convert to box-space
    float3 positionLS = mul(float4(cubemapWorldPos, 1), obbMatrix);
    float3 rayLS = mul(reflectVec, (float3x3)obbMatrix);

    // Original implementation used -1.0 - 1.0 space, we use 0.0 - 1.0 instead
    float3 firstPlaneIntersect  = (float3(1.0f, 1.0f, 1.0f) - positionLS) / rayLS;
    float3 secondPlaneIntersect = (-positionLS) / rayLS;
    float3 furthestPlane = max(firstPlaneIntersect, secondPlaneIntersect);
    float dist = min(furthestPlane.x, min(furthestPlane.y, furthestPlane.z));

    // Get the intersection position in worldspace
    float3 intersectPositionWS = cubemapWorldPos + reflectVec * dist;
    // Get corrected reflection direction
    return intersectPositionWS - cubemapPos;
}
```

Various limitations were hit along the way, however. Due to the wide range of hardware it operates on, a number of shaders have variants compiled for different *Shader Model* versions, often ps20, ps20b and ps30 (the last version fully supported by the DirectX 9 API). Advanced shading features added to the engine often had to be stripped out due to shader instruction caps or other limitations, and as such, parallax correction will only be added to the ps30 variants of each important game shader; with the exception of LightmappedGeneric which has enough unused instructions to allow for parallax correction on ps20b. The most important shaders, and the ones that make up the majority of the game world, are as follows:

Source Engine Common Shaders	
Baked Cubemaps	*Cubemap textures assigned during map compile process*
lightmappedgeneric_ps20/ps20b/ps30	Common brush shader. Supports a wide variety of layers, including detail textures, normal maps, and texture blending, as well as precalculated radiosity lightmaps
refract_ps20/ps20b/ps30	Effect shader. Distorts the rendered scene using a cubemap. Uses baked cubemaps when applied to brush entities.
Dynamically Assigned Local Cubemaps	*Cubemap textures assigned at runtime based on lighting origin in world-space*
vertex_and_unlit_generic_ps20/ps20b/ps30	Common model shader. Uses a lambert (or half-lambert) shading model with vertex lighting, or unlit. Supports detail textures.
vertex_and_unlit_generic_bump_ps20/ps20b/ps30	Common model shader. Similar to above, but supports bump maps and extra detail types. Uses a lambert (or half-lambert) shading model with vertex lighting, or unlit.
phong_ps20b/ps30	Advanced model shader. Uses Phong shading model.
refract_ps20/ps20b/ps30	Effect shader. Distorts the rendered scene using a cubemap. Uses dynamic cubemaps when applied to model entities.
Planar Reflections	*Uses planar render texture reflection. Cubemaps assigned during map compile as fall-back*
water_ps20/ps30	Water surfaces. Supports flowmaps to modify movement of bump and detail map, as well as precalculated radiosity lightmaps. Cubemaps can be used for a Fresnel effect or as a fall-back on low-end systems.

- will have parallax correction applied *- will not have parallax correction applied*

Parallax correction has been implemented in the majority of common shaders seen within Source, including those used for most world geometry and models within the world. However, it has not been added to the shader used for Water, as this already features high quality dynamic planar reflections, and simply uses cubemaps for minor supplemental Fresnel effects (or as a fall-back on low-end systems, where parallax correction is disabled anyway). Adding parallax correction here will add complexity to an already expensive shader with little visual improvement.

Shader source code for Source contains a header defining named options with ranges of numerical values, marked as static or dynamic. These are similar to C pre-processor defines, and each combination of these are compiled against each other ahead of time, resulting in some shaders with millions of combinations, to avoid the speed detriment of branching statements on older GPUs and to cut down on the time that it would take to generate these shaders on players systems at runtime. Combos can also be marked as skippable, avoiding compilation of unnecessary combinations; in this

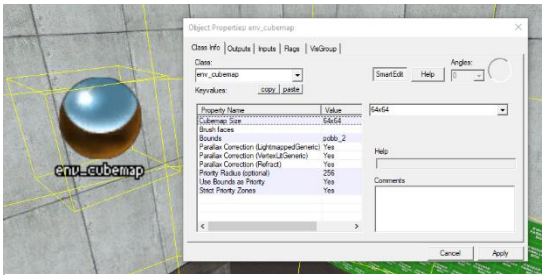
case, all shaders will skip their parallax correction flag if the cubemap flag is false, as there is no use in correcting nothing.

Shaders at runtime are made up of several parts; the compiled vertex and pixel shader files, which contain the GPU-side code compiled using the platform-specific shader compile tool (in this case, Microsoft's *FXC* tool), and the CPU-side shader DLL which sets everything up for the draw by setting constants, mesh data and textures to appropriate memory locations and telling the shader API to perform the draw.

The CPU-side shader DLL has multiple sections for each shader, but it largely boils down to two states; the *shadow state* and *dynamic state*. The shadow state is called once per material, and effectively acts as the static set-up pass for parameters that are unlikely to change. Should changes occur, the potentially expensive set up process will be repeated. The dynamic state is called before each render pass, allowing modification of dynamic values. In LightmappedGeneric, parallax correction will be a static option, as these values are baked into the material patches ahead of time. In every other shader, parallax correction will be controlled in the dynamic state.

Editor Integration

A fundamental aspect of parallax corrected cubemaps is the proxy geometry to correct the cubemap against. In this case, the object-aligned bounding box was chosen, as it offers enough versatility for the majority of game environments seen in Portal: Outside Influence.

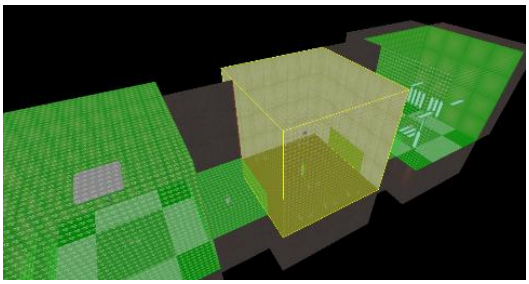


An env_cubemap entity, which is used as a sample point, and contains additional detail

To tie in with the existing world authoring process of Source, parallax correction volumes exist as *brush entities* (brush meaning authored world geometry) with the entity definition of *cubemap_obb*. These have no data besides a name and the brush geometry itself. Cubemap sample points, defined as *env_cubemap* entities, have a target attribute which can be set to the OBB entities. A number of other attributes can be set here too, including cubemap image resolution, types of entity to apply to (this allows a level author to disable parallax correction for dynamic entities or brushes, should a situation arise where correction is desired on one but not the other), and priority settings. These priority settings have been introduced to allow level designers to have stricter control over how entities choose their local cubemap, including offering the capability of using a radius or even the parallax OBB as an influence volume.

Matrix Calculation

In order to efficiently perform an intersection test against an object-oriented bounding box, an inverse transformation matrix needs to be calculated. This is performed during the map compile process, as this data does not change at runtime. While Hammer has built-in functionality for retrieving the bounds of a given object, these bounds are axis-aligned. As such, more complex process for calculating the object-oriented bounds of a brush was required.



Multiple cubemap_obb volume entities, tied to respective env_cubemap entities

Brushes are made up of a set of planes known as *windings*. Each plane is clipped against every other plane to form a convex shape that is then rendered. While loaded into the Hammer Editor, windings also have other details that help in the conversion process, including the points of the face in a clockwise order, as these are used to triangulate the winding for rendering on the GPU. Using these points, a transformation matrix for the OBB can be calculated.

Starting from the first winding, its first point is saved as the 'corner' of the shape, acting as an origin for the rest of the calculations. Local axes Y and Z are obtained from the second and fourth winding points respectively, with the third local axis, X, being more complex to calculate. First, a value (T) is obtained by taking the cross product of Y and Z. This T is then multiplied by the absolute cross product of the diagonal length of the third winding and T to finalize the X axis. The corner and axis values are fed into a 3x4 transformation matrix with X, Y, Z and corner taking up the columns in order. This matrix is then inverted to generate the matrix needed by the shader in order to perform fast and cheap ray-intersections in local-space to the volume. Effectively, this process simply calculates the length, height and width of the volume in the appropriate axes and uses a corner as the origin; the use of the corner origin is the reason for the change from -1.0-1.0 to 0.0-1.0 bounds in the shader as mentioned previously.

Baking into World Brushes

By far the most straight-forward part of this implementation was in adding this technique to world geometry. The render and asset pipelines in Source were written to be as fast as possible at runtime, and meaning that large amounts of data are precalculated and baked where ever possible. This is the same for cubemaps applied to brush surfaces. Each reflective material in the world has a 'patch' applied within the map file, binding the specific local cubemap and other required data to that particular surface. Adding the calculated parallax OBB transformation matrix to this match material directly passes this data into the shader system without going through any extra steps. Hooking into this step was the fastest and most reliable method of passing the required parallax data through to the *LightmappedGeneric* shader utilized by most geometry; however, this is not a viable solution for objects that have their local cubemap set or modified at runtime.

Dynamically Selected Local Cubemaps

While the brush objects that make up the majority of the static environment in Source feature baked cubemap samples, dynamic objects, such as props and NPCs, have their cubemaps set at runtime by locating their closest local cubemap sample point. As a result, their parallax matrices will change with these samples, preventing them from being baked in the same manner as brushes. Two attempts were made at implementing this feature, one requiring modification of the render pipeline, the other requiring a change within the texture management system.

Attempt #1: Modifying the Pipeline

The Source Engine is relatively modular, splitting various sections of the engine between multiple DLLs (Dynamic Link Libraries). This allows developers to switch out or update various components without needing to update every single one, but introduces some challenges. Perhaps the biggest challenge in implementing the parallax correction feature has been in getting the parallax data from the map loader found within the engine DLL, via the client DLL, to the DX9 shader DLL.

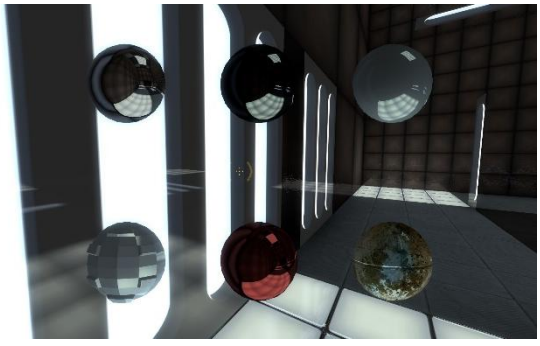
Retrieving the required details from the baked patch materials used by brush surfaces was deemed too awkward a method, and would have required looping through every patched material in the

map in order to gather data. Instead, another addition to the level compiler process was added within the VBSP *(Binary Space Partitioning)* step. During the cubemap saving process, a set of extra *'VPD'* data files, authored in the *Valve Keyvalues2* format, are added to the level package, which are then read during the level load process. This extra data is tied to the cubemap sample itself rather than being patched on a brush surface, and allows for easy retrieval of the parallax matrices and other information that may be needed, such as the priority zone rules for each sample.

The rules for assigning local cubemaps were adjusted to read from this data and take it into account when deciding which candidate cubemap sample to apply to an object. In Source's default implementation, the engine simply tries to find the closest cubemap sample to the lighting origin of point a given object. In the new implementation, spherical and box volumes must be considered a priority, and some cubemaps can specifically require being within one of these volumes to be applied at all, usually in an attempt to prevent the warping that can occur by leaving a parallax volume. The new process involves checking for the closest samples, but excludes those explicitly requiring being within a volume if that check fails, and forces the closest sample to be overridden if within one of those volumes.

Now that an appropriate sample has been collected, the problem of how to get the data from the engine into the renderer became the focus of attention. In Source, the assignment of local cubemaps is handled by the same light-cache system used to set the light values on a given object. A light-cache entry for a given position is requested by the render pipeline when calculating render attributes for a model, and a data object is either generated or retrieved that corresponds to that point in the world, including lighting data and local cubemap (as an *ITexture* interface pointer). This lighting data object is then passed through the renderer, its data being split out into other structures and functions, with different handling and caching for static prop models and dynamic ones. While it was not intended to drastically modify this pipeline, additional data was needed to be passed along, and the fastest solution was seen to simply pass the required matrix and positional data alongside every use of the cubemap texture pointer.

While adding the various required function overloads was a slow and tedious process, the first major challenge came about when, after drilling down through the render pipeline into the material system, the cubemap texture was handed off to a render context requested from the ShaderAPI DLL. This is where the cubemap is actually being assigned for access in the shader helpers and renderer itself. Adding getter and setter calls to the render context interface and its respective implementation class, it was finally possible to access the parallax data from within a shader. Modifying the render context interface introduced new issues, crashes that could only be fixed by rebuilding libraries and tools that were dependent on the render context, including the GUI framework, Hammer World Editor, and HL Model Viewer; some of which did not have available source code. This issue could be worked around by keeping tools in a separate binary directory to the game files, and all required libraries for the game itself could be recompiled with no issue.



Parallax correction drawn on LightmappedGeneric, Refract and VertexLitGeneric shaders in-engine

The second real issue in this implementation attempt came from the queued material system used by the threaded renderer. Due to its age, Source is fairly reliant on single-thread performance, even on modern systems. However, various attempts were made to make use of the many threads now

available on modern systems, including the implementation of a threaded render system. This was not only added to take advantage of modern PC systems, but to aid performance on game console hardware of the time, and allow for enhanced draw call batching. However, simply storing the parallax values in a naive manner as before resulted in every object in the world sharing the parallax information of the most recently assigned local cubemap, rather than their own appropriate local cubemap details. The code was then switched from passing the matrix values through directly, and instead passing through a pointer to the parallax data objects themselves. Once the appropriate calls were duplicated for the queued version of the material system, this issue was considered corrected.

Unfortunately, after stress-testing in more complex environments, the issue was not fixed for all objects in the world, only dynamic ones. Static props making use of the static light-cache were still having their cubemap parallax data assigned when they shouldn't, meaning that somewhere in the renderer, something was still not being correctly cached or set up before each draw call. An even more pressing issue was the incompatibility various aspects of the game and toolset now had with the modified render context. While this issue could be worked around with the Model Viewer, and some tools could be recompiled against the new context, the *Puzzlemaker* level editor feature of Portal 2 (a feature that is to be carried over into Outside Influence) could not be recompiled as Valve had been unable to provide source code for it, only a precompiled static library and header files. As a result, this attempt at implementing parallax correction in game entities was considered a dead-end. While a new interface could be added to the underlying render context offering support for loading of the parallax data, it was impossible to be certain that the Puzzlemaker did not rely on the exact layout of the underlying implementation class (possibly bypassing the interface), and the torturously long number of steps that had been modified within the light-cache system made this method difficult to maintain and debug. These changes were rolled back, and a new method was sought.

Attempt #2: A New Texture Type

So, after the relative failure of the previous attempt, a different tactic was required. Luckily, some of the early work, such as reading the parallax data into a structure upon map load, was still perfectly sound and required no modification. To reiterate, the core problem is in moving additional data alongside the cubemap texture and getting it to the shader DLL. The ITexture interface class, which is returned by the material system DLL upon loading a texture from the game files, revealed little in the way of built-in methods for supplemental data within the texture. However, this led to a new line of thinking; if it isn't viable to pass the data manually alongside the texture, then the data should be packaged along with the texture, and as ITexture is an interface to an unknown internal implementation, it should be possible to add a new internal texture class that stores the data required.

And so, that is what was attempted next. A new texture class, *CCubemapTexture*, was implemented within the material system DLL, inheriting from the standard internal *CTexture* implementation behind the ITexture interface, as well as a new interface called *ICubemapTexture*. This new interface allows for getting and setting of appropriate parallax related data, including the world position of the sample point, the transform matrix of its parallax correction OBB, and which types of object it should apply to. The new class also offers functionality for checking whether a point is within its priority volume, simplifying the local cubemap selection function discussed earlier.

In order to be sure of maintaining compatibility with all existing applications and libraries making use of the material system interface, textures of this new type can be requested by using the *nAdditionalCreationFlags* attribute of the material system's *FindTexture* function, instead of

implementing an additional function for getting these special textures. Valid cubemap textures can be cast to the ICubemapTexture throughout the application to access the new functionality, without the pipeline even being aware that it is not pointing to a standard CTexture object thanks to everything being hidden behind the ITexture interface. The new cubemap texture type makes it all the way through to the shader DLL, passing through the engine, game client, and ShaderAPI DLLs with no issue, and with no modification to the actual pipeline once it has been loaded in. This completely avoids the issue of trying to find the many locations where a cubemap texture may be cached between renders.

Per-Instance Command Buffers

Sadly, even after changing how information was being passed around, there arose another problem. Just as before, some objects were having their parallax matrices switched to the most recently applied parallax matrix instead of their local one. Stranger still, some shaders weren't even able to access their local cubemap from the shader helper class at all during the dynamic phase of the render. This led to continued research into the steps cubemap textures take through the render pipeline, revealing a step that not been considered: cubemaps being assigned through per-instance command buffers, and not in the dynamic shader setup phase. The reason for this is simple; static models in the world do not regularly call into the dynamic setup state, as their position never changes, and will only enter the dynamic state if the environment around them changes in some way (such as a flashlight being cast upon them).

Per-instance command buffers are the fastest method of modifying the parameters of a pass in the Source render pipeline, but has severe limitations, with shader authors being limited to a specific set of pre-existing commands prebuilt into the ShaderAPI DLL instead of being able to add custom ones. In order to pass the parallax correction data off to the shader on a per-instance basis, a new per-instance command would need to be added, which could then be added to all the required shader classes. Luckily, this required simply adding the new command to the tail of a switch statement and adding the required shader constant installation code. This is a slightly strange design choice, restricting those using the public Source SDK from adding their own command types as they do not have the ability to modify the ShaderAPI DLL, but is not an issue here.

This led to yet another issue. Some cubemaps may wish to still retain their infinite parallax (such as for large outdoor environments, or locations where correction volumes are completely unable to fit). Previously this was handled in the dynamic state in the shader DLL, but the use of the command buffer prevents us from changing these dynamic state values. In a slightly awkward workaround, the unused 4th field of the position vector of the cubemap sample coordinate passed to the shader was used to tell if correction should be performed, with a corresponding if statement in the shader skipping the correction should it be disabled.

With this final change, and a compilation pass of all modified shaders, parallax correction against OBB proxy geometry is fully functioning for both world geometry and game entities.

Accounting for Volume Edges

One of the drawbacks of the parallax correction technique is the warping and distortion that can occur when objects leave the proxy geometry being used to test against. While level designers should do their best to avoid this when adding correction volumes, issues still occur when at the boundary between two regions due to the lack of per-pixel cubemap blending in this

implementation. In testing, it was most apparent with the player's first-person view model, which is always in front of the camera in world-space but drawn after the rest of the game, resulting in an object outside of the volume should players get too close to the walls of a room. Some attempts have been made to mitigate this, to varying degrees of success. In order to solve the view model issue, the position of objects in the volume's local space are clamped to avoid ever hitting the extreme edge values. Once an object passes outside of this volume, the image of the reflection is faded to black over a short distance, the thought being that no specular reflection at all would look less jarring than the distorted, warped reflection or switching back to an uncorrected one.

Warping seen when leaving the parallax correction volume (left, inside the volume; mid, half inside the volume; right, leaving the volume)

Performance Statistics

Authoring Speed Impact

During the creation of a game environment, consideration must be taken in many aspects, including the position of light and reflection sources. In the case of parallax correction volumes, thanks to leveraging the Hammer Editor's built-in support for brush geometry editing, it is relatively easy to add required parallax volumes and takes only marginally longer during the art-pass of an environment to set up with major visual benefit. There is no way to view a live preview of this reflection, as the aging Hammer editor lacks any capacity for previewing baked lighting or reflection data, requiring the environment artist to run the map to see changes. Realistically though, the artist should be fitting the volumes to the given environment and will likely not face many issues wherein placement of these volumes causes issue. It is worth noting that these volumes are not mandatory, and cubemap samples will simply fall-back to the previous infinite parallax implementation if one is not defined.

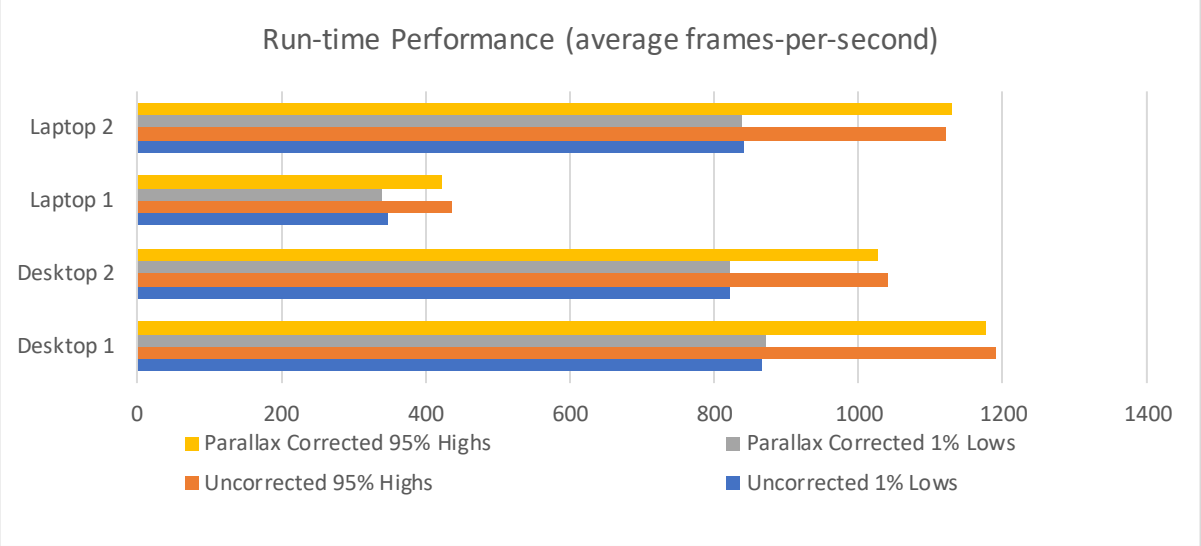
Build-time Performance

The VBSP step in which cubemap data is calculated is virtually unaffected by this change. In essence, it boils down to writing a very small amount of additional data to the level file for each sample point, and as the VBSP step was already the fastest part of the level compile process, the changes introduced have made no meaningful impact on the compile time or level size. Compilation time of the VBSP step before these additions measured at an average 1.47 seconds, and after measured in at an average 1.46 seconds – the difference is simply a matter of run-by-run variance and falls within an acceptable margin of error. These tests were run 10 times each, with the mean average calculated and used as the accepted value.

The cubemap images themselves are captured with the *buildcubemaps* command in the engine itself. This has not been modified in any way and thus has no difference in performance while generating.

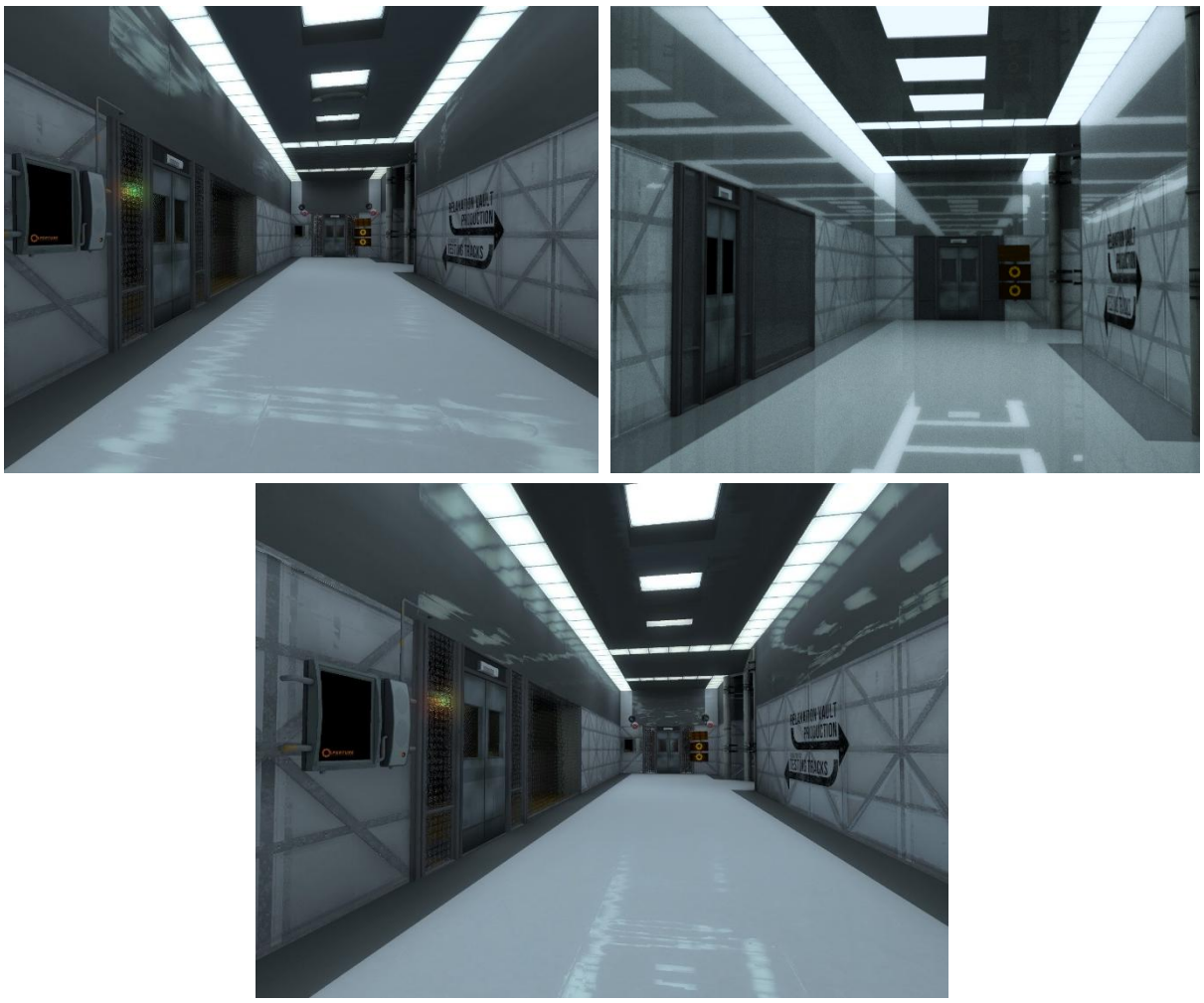
Run-time Performance

With very minimal extra data to load in from the file during the map load phase, engine load times are virtually identical to previously. Requesting the CCubemapTexture internal type takes no additional time as requesting the CTexture internal type, and installing the required data is a matter of reading in 5 sets of 4 floating point values from a text file and applying them via the ICubemapTexture interface methods.



The final implementation of OBB corrected reflections has had no practical impact on performance on any of the hardware tested on, *(see Appendix B for specifications)* with framerates far above the 60FPS stable minimum. While this is partially due to the parallax correction implementation being relatively lightweight, it largely comes down to the relative simplicity of the shaders in Source, and the fairly low-level approach to handling them, when compared to more modern game engines such as Unity 3D or Unreal 4. Performance was measured in the same test environment over the course of 10 minutes, taking a weighted average of the highest 95% of framerates recorded and the lowest 1%. Performance variance between corrected and uncorrected is so minor, with framerates still recorded above 1,000FPS on multiple pieces of test hardware, that it is expected all players will be able to have the feature enabled while maintaining the minimum target framerate of 60FPS.

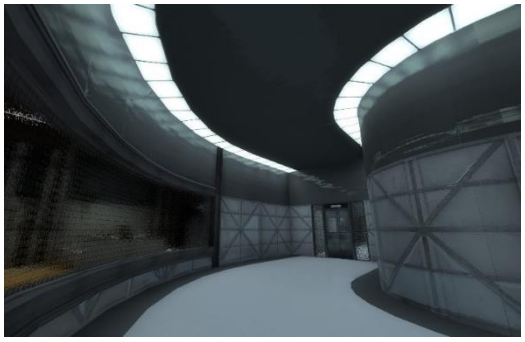
Comparison with Visual Target



Uncorrected cubemaps in-engine (top-left), the 3DS Max ray-traced visual target (top-right), implemented parallax correction in-engine (bottom)

Comparisons with the initially produced visual target are positive, with a noticeable improvement in reflection accuracy when compared to the earlier cubemapping implementation captured from the same location in the same scene. The visual target shows a larger amount of detail than the final implementation, however the ray-traced visual target did not make use of normal mapping in the same way as the Source Engine's shaders do for perturbing the reflection vector, and the scene in-engine is making use of only a 64x64 pixel cubemap sample in order to save on graphics memory and storage space.

While it was considered that this implementation may lead to issues in some complex environments, some convincing reflections have been authored in more geometrically complex environments too through clever use of multiple sample points and correction volumes, demonstrating the versatility of the effect. With minor exceptions, this implementation could suit all the planned game environments in Outside Influence.



A curved corridor with convincing reflection using 3 cubemap samples and OBB volumes

Parallax Correction with Complex Geometry – Possible on DirectX 9?

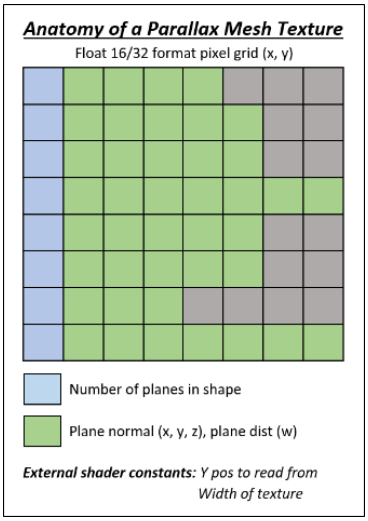
Background

As mentioned previously, when Sébastien Lagarde published their work on parallax correction of cubemaps, they mentioned the use of more complex proxy geometry than a box or sphere, discussing how they worked with convex geometry in the production of Remember Me. The details on this iteration were sparse, and this more complex correction has not been attempted in many titles. After getting in touch for more details, they stated that it's possible with any geometry that an intersection test can be performed against, and called in Krzysztof Narkowicz to back up their suggestion. However, it was then stated that it is likely that none of this is possible under DirectX 9.

'Hardware Morph' facial animation in Source make use of a texture to stream large amounts of data to the vertex shader (Valve Corporation, 2007)

After implementing the OBB method of parallax correction, thoughts again turned to this idea; what barriers were actually in place preventing use of this method? The biggest obstacles are in passing the mesh data to the shader to perform the intersection test, and the complexity of the intersection test itself. In the DirectX 10 API, large amounts of additional data could be passed around via the use of *constant buffers,* which allow for up to 4096 *float4* vectors to be passed into shaders¸ with this capability only growing with newer iterations of the API. Some research led to a thread on StackOverflow wherein a user was trying to pass large amounts of data to a shader in the *OpenGL* API. While the minutia of this thread are not entirely relevant, one idea that certainly was involved passing the data through as a generated texture, using a float-based image format such as *GL_R32F*. Searching through the Source engine's codebase for references to the DirectX 4 value equivalent, *D3DFMT_A32B32G32R32F,* and found that Valve are actually using this technique in various places, such as the *hardware morph (HWM)* system used for high-quality facial animations.

Prototype Implementation



First, it is prudent to refer back to advice given by Krzysztof Narkowicz when inquiring about the parallax correction technique; *"Any convex is just a test against all it's planes. Also BSP could be directly used to raytrayce against to get best env reflection, albeit most likely not on DX9 :)".* (Narkowicz, 2018) While it is untenable to compress the entire map into a texture and test against those planes, it may be viable to send the planes of each winding of the correction brush and test against those. Plane-intersection tests are relatively cheap, especially if a relatively low face count is used. However, the use of this requires correction volumes to be convex; but this will not commonly be an issue as the cubemap image is not able to contain both the back and front faces of any given shape due to being sampled from a single location. By packing all of the data into a single texture for the entire map, it avoids needing to load a separate texture for each cubemap sample through another per-instance command buffer entry. Even with the maximum cubemap count of a map, 2048 samples, this is still a viable option instead

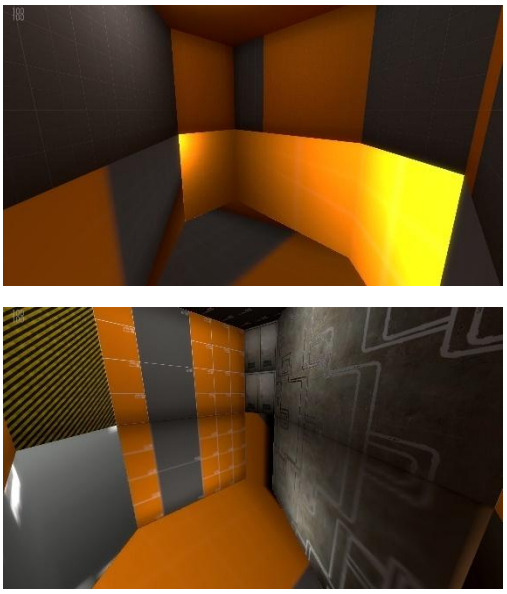
of repeatedly switching the texture in and out. In fact, this texture could be even more tightly packed with a more mature data fetch routine.

An initial prototype was created, first by adding another step to the map compiler to calculate the planes of a brush face (defined by its normal and distance from the origin) and write them to a 32-bit floating point texture file, which is then bundled into the map file. Each cubemap has a unique ID assigned that corresponds to a Y position in the texture, with the first coordinate in the X direction describing how many planes the shape contains, to allow sampling of specific unfiltered pixels from the texture without use of the HLSL *Load* function available in DirectX 10. This advanced plane method of correction is only used when specifically requested, or a correction shape has more or less than 6 sides, as the standard box correction technique is computationally cheaper and does not require looping. The amended correction method and intersection function can be seen here:

```
bool IntersectRayPlane(float3 rayOrigin, float3 rayDirection, float3 planePosition, float3 planeNormal,
out float3 intersectionPoint) {
    float rDotn = dot(rayDirection, planeNormal);
    if (rDotn < 0.0000001f ) {
        return false; // no intersection occurred; facing away or parallel
    } else {
        float s = dot(planeNormal, (planePosition - rayOrigin)) / rDotn;
        intersectionPoint = rayOrigin + s * rayDirection;
        return true;
    }
}

(...)

// Find the closest intersection to the camera
float xSampleDist = 1.0f / (g_MeshDetails.x);
float ySamplePos = g_MeshDetails.y;
int numPlanes = tex2D(MeshSampler, float2(0.0f, 0.0f)).x;
float closestDist = 999999999999.0f; // Stupidly high distance
float3 closestIntersection = float3(0.0f, 0.0f, 0.0f);
for(int j = 0; j < 32; j++) { // max planes = 32
    if(numPlanes > j) { // exit condition
        float4 plane = tex2D(MeshSampler, float2((j + 1) * xSampleDist, ySamplePos)).xyzw;
        float3 intersect = float3(0.0f, 0.0f, 0.0f);
        if(IntersectRayPlane(worldPos, reflectVect, plane.xyz * plane.w, plane.xyz, intersect)) {
            // If current intersection is closer
            float dist = length(intersect - worldPos);
            if(dist < closestDist) {
                closestIntersection = intersect;
                closestDist = dist;
            }
        }
    }
}

reflectVect = closestIntersection - cubemapPos;
```

This was added to the LightmappedGeneric shader in order to test, and while the correction does appear to be functioning somewhat correctly, with the shape matching the room as it should, the reflection vector appears to be entirely incorrect, albeit in a stable fashion with no warping when changing camera position. It hasn't been worked out exactly what is going on here, but it is clear that while some adjustment needs to be made, this technique is definitely viable under DirectX 9. There is also a measure of awkwardness to the code here; the loop has a strict limited size, with the intersection only being called when data is available – this is due to a bug in the DirectX 9 shader compiler which will consider all loops as infinite if comparing an iterator to external constants, even if that constant has its value clamped within the very same function.

In fact, it should be theoretically possible to test against concave geometry too, providing a higher degree of flexibility to level authors. Switching out the current method of saving brush winding planes, instead the brushes (multiple would be required for concave geometry as brushes themselves must always be convex) need to be converted into a triangulated mesh before being saved to the texture. Luckily, Source already has an open-source map-to-model converter, known as *Propper*, which will convert level geometry into a standard polygonal model for export. (Foust, 2009) ('tuxxi', 2016) Adapting the triangulation functionality of this tool, and stripping out unneeded capabilities such as texture mapping, it is relatively simple to produce a system in which a brush could be converted from a set of windings to a triangulated mesh. With multiple brushes tied to the correction entity, and faces with the *'tools/toolsskip'* material being skipped to cull unwanted or interior triangles, it is possible to make concave correction volumes that accurately matches the level environment. All of this can be written to a texture as before, and have ray-triangle intersection tests (McGuire, 2018) performed against it. These tests are more complex than the plane tests performed against convex geometry. This raytracing is likely very slow as current graphics hardware is not optimised for the static-branching seen in these algorithms, but given the limited number of triangles that will be tested against, it is not beyond the realm of possibility. It may in fact be best to switch between these methods on the fly based on the number of brushes tied to a correction entity, the number of valid sides on the entity, and whether the final shape is convex or concave.



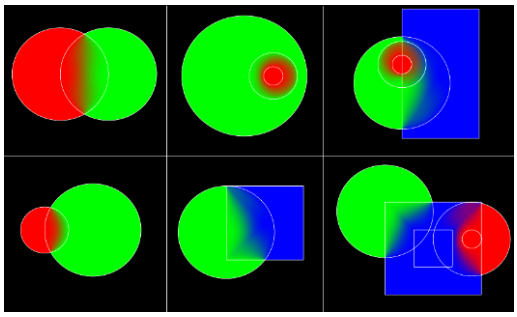
Plane-based convex shape correction tests. Certain walls feature entirely incorrect reflections, due to a yet undetermined bug

This prototype implementation proves that this technique is entirely possible under DirectX 9, albeit with continued work, and even performs similarly to the OBB method, again likely owing to how few planes we are testing against and how old the engine and graphics API are, as well as the relative simplicity of the shader compared to modern titles.

Conclusion

Future Research

While the implemented OBB-based correction feature solves the parallax problem in many cases, it is not a best-fit for every case. Beyond completing the implementation of the convex-shape method prototyped above, a concave shape version could be created to offer even greater flexibility. The convex-shape method is relatively close to completion, with some surfaces exhibiting correct reflection and others not, as seen here. Though some brief continued experimenting, it is currently thought that the planes themselves are either not being passed through correctly, or not being created correctly. Due to the relatively small numbers involved in plane normals, it is possible that somewhere something is being munged, either when generating the texture or when reading it.



*Debug view of cubemap influence zones technique from **Remember Me*** (Lagarde & Zanuttini, 2012)

Once an increased complexity of shape is implemented, it would be prudent to look at other aspects of the reflection system and how they could be improved. While out of scope for this implementation, which strictly focussed on parallax correcting the cubemap image, the largest improvement in reflection rendering would likely be found by implementing a blend between multiple cubemap images, as seen in Remember Me, a source of much inspiration for the work produced here. This could be achieved in various ways, such as by reworking the pipeline to pass a set of cubemap samples from a point instead of the single acquired sample; but is perhaps best achieved by switching to a deferred render pass for cubemap reflections. Influence volumes would be used to dictate where cubemaps should apply with gradient falloff, similar to how light is handled in many cases. This was specifically avoided in this project, as it was part of the initial plan to modify as little of the overall pipeline as possible to achieve the effect, however it would allow for per-pixel application of static reflections, complete with proper parallax correction. Such a change would require the enabling of the g-buffer and appropriate shader changes. Luckily, a project known as *Source Deferred*, exists to bring a fully deferred render pipeline to Source. By adapting work from Source Deferred, it may be possible to improve the rendering of many aspects of the game while implementing the desired per-pixel static reflection.

Beyond this, implementation of a screen-space reflection technique, perhaps only rendering dynamic models, would be a great addition to the rendering. With an existing and functional parallax correction technique to fall-back to, the implementation of an additional layer of dynamic reflections could tie the game world together in a visually pleasing way. This application of dual techniques has been seen in many recent titles, such as Rainbow Six: Siege; the corrected cubemap does a convincing job of filling in the gaps in the available screen-space data, making for a less jarring drop-off in the reflected image.

*Interiors of inaccessible rooms in buildings in **Spider-Man** are faked via a parallax correction technique* (Insomniac Games, 2018)

An interesting use for parallax corrected cubemapping is application beyond specular reflections. Effectively, the technique allows for environments to be faked from a static image without the use of real polygonal geometry at runtime. This has many uses, but has been seen in recent titles *Overwatch* and *Spider-Man* to add interiors to buildings without requiring artists to produce a unique interior and without requiring players to actually render the room. This is especially important in the latter, as rendering every outward-facing interior room of a tall building or skyscraper on console hardware is impractical, especially while also needing to draw a geometrically complex city with simulated inhabitants. The effect is convincing unless studied directly, and adds a sense of depth to the game environment that would be lost by simply adding darkened glass squares to the buildings as in earlier titles. Some early tests have been conducted in Outside Influence to simulate an observation room in a similar manner, by using a parallax correction volume outside the bounds of the world and a pre-rendered image of a fully modelled observation room. Much like in Spider-Man, the effect is rather convincing unless viewed up close, and could be used in many cases as an alternative to building and rendering individual observation room environments.

*A test of interior mapping using the parallax correction implementation in **Outside Influence***

With the major improvement in specular reflections, it would be possible to use the same technology to improve diffuse reflection too, by adding a pre-blurred copy of each cubemap sample to use as an image-based diffused lighting source. The visual improvement could be debated however, as Source already features an effective baked radiosity simulation of global illumination that offers a pleasing visual appearance. Either way, this could be added with relative ease, and could eventually lead to the implementation of a full physically based rendering system, although the move to such a system would require rework of the majority of existing assets.

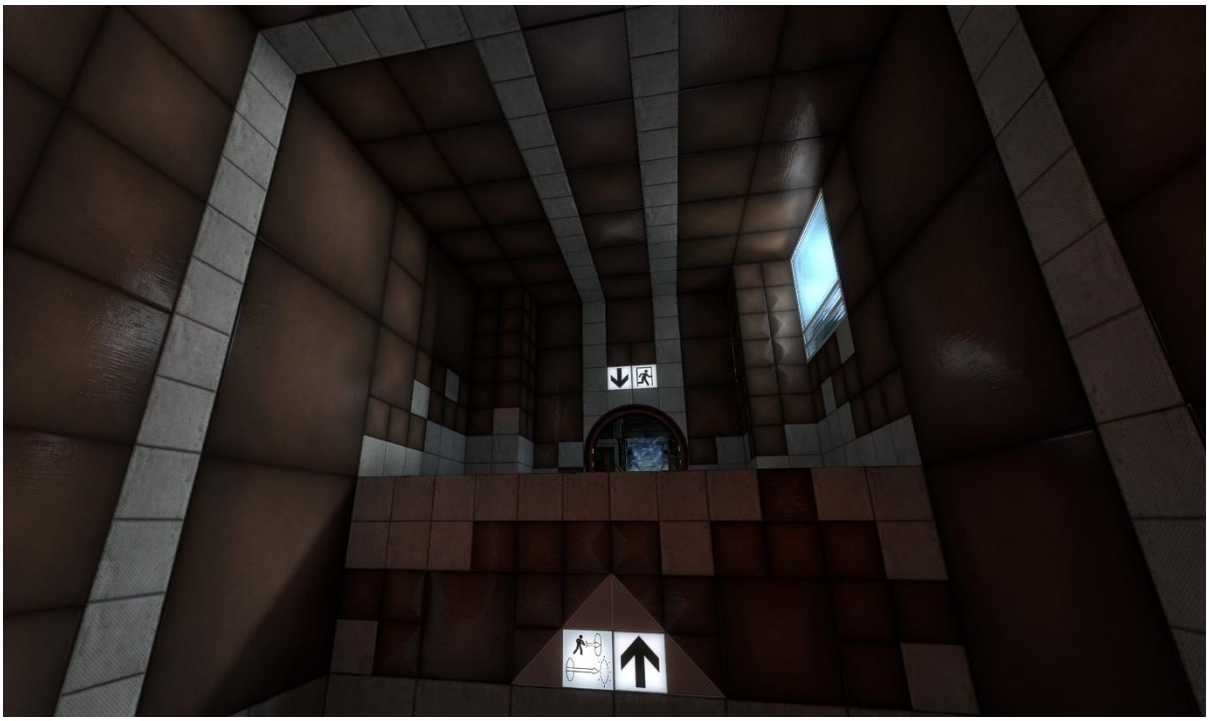
A final point to consider for the future is that the parallax correction effect is only currently supported on systems that can make use of Shader Model 3.0. While this isn't an issue hardware-wise – the *Steam Hardware Survey* states that only 0.82% of Steam users have cards that don't adhere to the Shader Model 4.0 spec (Valve Corporation, 2018) – this does raise problems with the two other launch platforms for the game, Mac and Linux. On these systems, DirectX calls are translated to appropriate OpenGL ones via Valve's ToGL translation layer, the public release of which supports up to Shader Model 2.0b. This has seemingly been addressed in Valve's more recent releases of Counter-Strike: Global Offensive, which make use of ps30 shaders on every platform. This may simply require Outside Influence to have its internal version of the ToGL translation layer upgraded, or may require considerable additional work. While definitely out of scope for now, it would be favourable to move from the DirectX 9 API to a newer, ideally multiplatform graphics API, such as DirectX 11 or Vulkan, to allow for better portability across platforms and offer new rendering functionality.

Final Observations

Overall, the result of this implementation has been a success, introducing far improved specular reflections to a game running on an older engine and limited graphics API. While certain limitations are in place from the original cubemapping system, such as a lack of support for blending between different cubemap samples, approximately spatially correct reflections can be achieved in the majority of the game with no measurable impact on performance, simply by placing correction volumes within the game world. The majority of Source's standard shader set fully support the feature, it can be added to others with relative ease if required thank to the use of a shared shader function, and game materials do not need to be modified to have newly corrected specular reflections applied. Parallax volume data is sent to the shader system by using a new internal texture type and interface class, allowing required information to be sent through the render pipeline without reworking the pipeline itself. Static environmental surfaces have their cubemap data baked at build-time to improve efficiency at run-time, while dynamic objects have their cubemap data set during run-time based on their origin position in world-space through the use of new per-instance command buffer functionality.

While the solution arrived at is limited to a box volume, this offers enough flexibility for the majority of environments in the game. However, various avenues could be taken in the future to expand support to convex geometry (as a set of planes) or concave mesh geometry (as triangles), by packing the required data to the pixel shader as a float-format texture; in theory allowing something on the older graphics API once considered impossible by a veteran graphics programmer.

The solution implemented here offers a balance between visual quality, speed and ease-of-use for environment artists, and drastically improves the rendering of specular reflections across the entirety of Portal: Outside Influence, bringing them far closer in line with modern standards without needing mass-rework of game assets.



Screenshot from Portal: Outside Influence, showing parallax corrected reflections (Elite Treacle Ltd., 2018)

Appendices

Appendix A: Volumetric Lighting and Portal Light Transmission

In the proposal for this dissertation, plans for two other modern graphical effects were laid out, these effects being '*volumetric lighting*' and '*transmission of dynamic light through portal entities*'. While it is strongly believed that both of these effects would add to the overall aesthetic of the title, the improved reflection rendering makes a major impact on the graphical fidelity of the environment and game entities. Not only this, but it does require some degree of additional work to be put into the level environments art pass while placing cubemap sample points. As such, it was deemed the more important of the three effects, as the volumetric lighting could be hooked into existing dynamic flashlight entities, and projection of dynamic flashlights through portals requires no specific level integration. No attempts were made of the other two effects beyond an initial batch of research into various methods, as the difficulties faced in passing parallax correction data through the engine was greater than initially expected. These effects may be attempted in the future to improve the rendering quality of Outside Influence even further.

Appendix B: Testing and Development Hardware Specifications

Desktop 1: Intel Core i7 6700k, 16GB DDR4 Memory, Nvidia GTX 980Ti 6GB

Desktop 2: Intel Core i5 8600k, 16GB DDR4 Memory, Nvidia GTX 1060 3GB

Laptop 1: Intel Core i7 3610qm, 8GB DDR3 Memory, Nvidia GTX 660M 2GB

Laptop 2: Intel Core i7 8750h, 16GB DDR4 Memory, Nvidia GTX 1070 8GB

Each implementation ran at speeds far beyond what was required, with even machine #3 reaching framerates above the engine frame-cap of 300FPS and far above the targeted 60FPS.

Appendix C: Image Gallery, Video Clips and Supplementary Material

Please the below link for high-resolution images, video clips and other supplementary material.
https://1drv.ms/f/c/daff8ed69ed9ed17/IgAp_aoYdsi5TILuuVlcLvYLAWuVu8_NqrDv4oyvbKG2Ee0

Appendix D: Acknowledgements

Thanks to Valve Corporation, Elite Treacle Ltd., LunchHouse Software LLC., Graham Dianaty, Brian Charles, Stephen Sperrin, Sébastien Lagarde, Krzysztof Narkowicz, Arienne Bloss, Simon Scarle, Gwenfrewi Haynes, Sophie Sharpe, and the staff at Reach Robotics.

References

Abrash, M., 1997. *Graphics Programming Black Book.* Special Edition ed. s.l.:Coriolis Group,U.S..

Apodaca, A. A. & Gritz, L., 1999. *Advanced RenderMan.* 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc..

Arvelius, J., 2005. *Wikimedia Commons: Reflection Angles.* [Online]
Available at: https://en.wikipedia.org/wiki/Reflection_(physics)#/media/File:Reflection_angles.svg
[Accessed 03 November 2018].

Autodesk, 2018. *3DS Max 2018,* San Rafael, California: Autodesk.

Bjorke, K., 2004. Chapter 19: Image-Based Lighting. In: 1st, ed. *GPU Gems.* s.l.:Addison-Wesley.

Bolas, N., 2013. *Stack Overflow: Passing a list of values to fragment shader.* [Online]
Available at: https://stackoverflow.com/a/7958008
[Accessed 16 November 2018].

Brennan, C., 2002. *Accurate Environment Mapped Reflections and Refractions by Adjusting for Object Distance,* s.l.: ATI Research.

Charles, B., 2013. *Parallax Corrected Cubemaps in the Source Engine.* [Online]
Available at: https://www.youtube.com/watch?v=ZH6s1hbwoQQ
[Accessed 20 November 2018].

Charles, B., 2015. *Parallax Correction for Source code samples - parallaxcubemaps.zip.* [Online]
Available at: http://13thsfg.us.to/files/parallaxcubemaps.zip
[Accessed 20 November 2018].

Courrèges, A., 2015. *GTA V - Graphics Study.* [Online]
Available at: http://www.adriancourreges.com/blog/2015/11/02/gta-v-graphics-study/
[Accessed 20 November 2018].

Czuba, B., 2010. *Box Projected Cubemap Environment Mapping.* [Online]
Available at: https://www.gamedev.net/forums/topic/568829-box-projected-cubemap-environment-mapping/
[Accessed 01 November 2018].

Dontnod Entertainment, 2013. *Remember Me,* Paris, France: Capcom.

EA DICE; Criterion Software, 2018. *Battlefield V,* s.l.: Electronic Arts.

Elite Treacle Ltd., 2018. *Portal: Outside Influence,* Bristol, UK: Elite Treacle Ltd..

Epic Games; Nvidia; ILMxLAB, 2018. *Star Wars: Reflections,* s.l.: Epic Games.

Epic Games, 2014. *Unreal 4,* Cary, North Carolina, US: Epic Games.

Epic Games, 2018. *Unreal 4 Documentation: Planar Reflections.* [Online]
Available at: https://docs.unrealengine.com/en-us/Engine/Rendering/LightingAndShadows/PlanarReflections
[Accessed 18 November 2018].

Flying Wild Hog, 2016. *Shadow Warrior 2,* Warsaw, Poland: Devolver Digital.

Foust, C., 2009. *Propper - Modified VBSP,* s.l.: s.n.

Fredriksson, J. & Scott, A., 2016. *Generating Real Time Reflections by Ray Tracing Approximate Geometry,* Gothenburg, Sweden: Chalmers University of Technology.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* Boston, US: Addison-Wesley.

GianniG46, 2010. *Wikimedia Commons: Lambert2.gif.* [Online]
Available at: https://commons.wikimedia.org/wiki/File:Lambert2.gif
[Accessed 2018 November 03].

Hammer, P., 2014. *The Rendering Technology of 'Lords of the Fallen'.* Paris, France, Game Connection Europe.

Hammerpoint Interactive, 2012. *Infestation: Survivor Stories (formerly The War Z),* s.l.: OP Productions.

Henrik~commonswiki, 2008. *Wikimedia Commons File:Ray trace diagram.svg.* [Online] Available at: https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg [Accessed 18 November 2018].

id Software, 1996. *idTech 2 Engine (Quake/Quake II Engine),* Richardson, Texas: id Software.

Immel, D. S., Cohen, M. F. & Greenberg, D. P., 1986. A Radiosity Method for Non-Diffuse Environments. *SIGGRAPH,* 20(4), pp. 133-142.

Insomniac Games, 2018. *Spider-Man,* Burbank, California: Sony Interactive Entertainment.

Kajiya, J. T., 1986. The Rendering Equation. *SIGGRAPH,* 20(4), pp. 143-150.

Lagarde, S., 2018. *Twitter: Tweet from Sébastien Lagarde to Sam Morris.* [Online] Available at: https://twitter.com/SebLagarde/status/1059465979351179264 [Accessed 15 November 2018].

Lagarde, S. & Zanuttini, A., 2012. *Local image-based lighting with parallax-corrected cubemaps.* Los Angeles, California, SIGGRAPH.

Liu, E. & Llamas, I., 2018. *Ray Tracing in Games with NVIDIA RTX (Presented by NVIDIA).* San Francisco, California, Game Developer's Conference.

Madsen, S., 2017. *How to Prioritize with the MoSCoW Technique.* [Online] Available at: https://www.projectmanager.com/training/prioritize-moscow-technique [Accessed 20 November 2018].

McGuire, M., 2018. Sample 2: Ray-Triangle Intersection. In: *The Graphics Codex.* s.l.:Casual Effects.

McReynolds, T. et al., 1997. *Programming with OpenGL: Advanced Rendering.* s.l., SIGGRAPH.

McTaggart, G., 2004. *Half-Life® 2 / Valve Source™ Shading,* s.l.: Valve Corporation/AMD.

Mendez, R. L., 2015. *Optimized Rendering Techniques Based on Local Cubemaps.* London, ARM Game Developer Day.

Narkowicz, K., 2017. *Rendering of Shadow Warrior 2.* Krakow, Poland, Digital Dragons.

Narkowicz, K., 2018. *Twitter: Tweet from Krzysztof Narkowicz to Sam Morris.* [Online] Available at: https://twitter.com/knarkowicz/status/1059476091180630016 [Accessed 15 November 2018].

Nvidia Corporation, 2018. *Battlefield V DXR Real-Time Ray Tracing Available Now.* [Online] Available at: https://www.nvidia.com/en-us/geforce/news/battlefield-v-rtx-ray-tracing-out-now/ [Accessed 18 November 2018].

Olson, S., 2012. *Wall Worm Model Tools,* s.l.: Wall Worm.

Reif, J. H., Tygar, J. D. & Yoshida, A., 1994. Computability and Complexity of Ray Tracing. *Discrete and Computational Geometry,* Volume 11, pp. 265-287.

Respawn Entertainment, 2014. *Titanfall,* Los Angeles, California: Electronic Arts.

Rockstar North, 2013. *Grand Theft Auto V,* Edinburgh, Scotland: Take-Two Interactive.

Sanglard, F., 2012. *DOOM3 SOURCE CODE REVIEW: DMAP (PART 2 OF 6) >>.* [Online] Available at: http://fabiensanglard.net/doom3/dmap.php [Accessed 12 November 2018].

Szirmay-Kalos, L., Aszódi, B., Lazányi, I. & Premecz, M., 2005. Approximate Ray-Tracing on the GPU with Distance Impostors. *EUROGRAPHICS,* 24(3).

Toft, A., 2017. *Fast, accurate reflections with parallax-corrected cubemaps,* Cambridge, England: Downing College.

tonfilm, 2013. *VVVV: Infinite ray intersects with infinite plane.* [Online] Available at: https://discourse.vvvv.org/t/infinite-ray-intersects-with-infinite-plane/10537 [Accessed 18 November 2018].

'tuxxi', G. u., 2016. *GitHub: Propper for Source SDK 2013.* [Online]
Available at: https://github.com/tuxxi/propper-2013
[Accessed 16 November 2018].

Ubisoft Montreal, 2015. *Tom Clancy's Rainbow Six Siege,* Montreal, Canada: Ubisoft.

Unity Technologies, 2005. *Unity 3D Engine,* San Francisco, US: Unity Technologies.

Valve Corporation, 1998. *Half-Life,* Bellevue, Washington: Valve Corporation (digital), Sierra (retail).

Valve Corporation, 2004. *Half-Life 2,* Bellevue, Washington: Valve Corporation (digital), Sierra (retail).

Valve Corporation, 2004. *Source Engine,* Bellevue, Washington: Valve Corporation.

Valve Corporation, 2005. *Half-Life 2: The Lost Coast,* Bellevue, Washington: Valve Corporation.

Valve Corporation, 2006. *Half-Life 2: Episode One,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2006. *Shader authoring/Anatomy of a Shader/Shader States.* [Online]
Available at:
https://developer.valvesoftware.com/wiki/Shader_authoring/Anatomy_of_a_Shader/Shader_States
[Accessed 15 November 2018].

Valve Corporation, 2007. *Portal,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2007. *Team Fortress 2,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2007. *The Orange Box,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2009. *Left 4 Dead 2,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2010. *Alien Swarm,* Bellevue, Washington: Valve Corporation.

Valve Corporation, 2011. *Portal 2,* Bellevue, Washington: Valve Corporation (digital), Electronic Arts (retail).

Valve Corporation, 2012. *Counter-Strike: Global Offensive,* Bellevue, Washington: Valve Corporation.

Valve Corporation, 2014. *GitHub: ValveSoftware/ToGL.* [Online]
Available at: https://github.com/ValveSoftware/ToGL
[Accessed 20 November 2018].

Valve Corporation, 2014. *Source Engine 2,* Bellevue, Washington: Valve Corporation.

Valve Corporation, 2018. *Steam Hardware and Software Survey: October 2018.* [Online]
Available at: https://store.steampowered.com/hwsurvey/
[Accessed 20 November 2018].

www.scratchapixel.com, n.d. *A Minimal Ray-Tracer: Rendering Simple Shapes (Sphere, Cube, Disk, Plane, etc.).* [Online]
Available at: https://www.scratchapixel.com/lessons/3d-basic-rendering/minimal-ray-tracer-rendering-simple-shapes/ray-plane-and-ray-disk-intersection
[Accessed 18 November 2018].

Yeunglm, S., 2011. *Spherical Harmonic Lighting.* [Online]
Available at: http://simonstechblog.blogspot.com/2011/12/spherical-harmonic-lighting.html
[Accessed 20 November 2018].