AUTODESK UNIVERSITY

DV21491

Tips and Tricks to Get the Most out of your Virtual-Reality Experiences in Stingray

Olivier Dionne – Autodesk Inc.
Benjamin Slapcoff – Autodesk Inc.
Andrew Grant – Autodesk Inc.

Learning Objectives

- Learn how to navigate VR devices in Stingray
- Learn how to build quick interactions using Flow nodes
- Understand and learn how to use the profiling tools in Stingray
- Learn how to use creative solutions to target best rendering quality under a tight performance budget

Description

Rendering high-performance virtual-reality (VR) experiences at required frame rates is complex and technically challenging. Creating memorable content and intuitive interactions forces developers to rethink old assumptions and come up with new tricks. The recent wave of VR applications showcases the opportunities and ingenuity of many developers to get around these hurdles. This class will focus on how to use Autodesk Stingray to create optimized VR experiences. By using the available profiling tools to spot bottlenecks, we'll share tips and tricks to tailor your content and interactions to get the most out of Stingray's VR rendering pipeline.

Your AU Expert(s)

Olivier Dionne is Development Manager of the Autodesk Stingray Rendering Team. Prior to joining Autodesk, he completed a B. Eng. (2004) in software engineering at Ecole Polytechnique (Montreal) and worked for various start-ups in the game industry developing high-performance real-time animation and rendering engines on embedded devices. After spending a couple years in production, he returned to Ecole Polytechnique to obtain his M. A. Sc. (2009), where he focused on realistic simulation of interactive soft tissue deformations for an interventional scoliosis surgery simulator at Saint-Justine Hospital (Montreal). His research interests include computer graphics, animation, image processing and geometric/physics-based deformation modeling.

olivier.dionne@autodesk.com Autodesk Inc.

Benjamin Slapcoff recently joined the Autodesk Stingray Rendering team full time after completing his B. Comp. Sc. degree at Concordia University, but he's been working on the team as an intern since 2014 on many different aspects of VR in the Stingray engine. He has prior experience in the game industry before Autodesk, working at middleware, start-up, and more established game companies since 2009.

benjamin.slapcoff@autodesk.com Autodesk Inc.

Andrew Grant is a Product Manager for Autodesk Stingray. Prior to joining Autodesk, he worked in game development at Monolith Productions, THQ, and Ubisoft. Over 16 years in the industry he has worked as a character artist, animator, and technical artist on games from No One Lives Forever, to F.E.A.R and HOMEFRONT, and many in between.

andrew.grant@autodesk.com Autodesk Inc.

Introduction

Autodesk Stingray is a modern game engine designed for architectural viz specialists as well as pro-indie game makers. Based on the core technology from the proven Bitsquid engine, Stingray strives to be an open and flexible interaction platform. It has been used for commercial games such as:

- Warhammer: End Times - Vermintide (PC, Xbox One, PS4)
- War of the Roses (PC, Online)
- Krater (PC, Mac)
- Hamilton's Great Adventure (PC, PS3, Android)
- Helldivers (PS3, PS4, PC)
- Gauntlet (PS4, PC)
- The Showdown Effect (PC)

Outside of games, Stingray is also driving the Autodesk Live Viewer, which immerses people into the story of an architectural design. With one click in Revit, Autodesk Live converts a project visualization into an interactive model in the cloud, which can then be downloaded for viewing in real-time on different platforms.

As illustrated by the few examples mentioned above, Stingray's modern real-time renderer can power a broad variety of games and applications ranging from third-person, top-down, 2.5D side-scroller, architecture visualizations, VR, simulations, etc., which all have very different needs with respect to rendering. To support wide-ranging requirements such as refresh rates (30-120Hz), artistic style, post-effects, shadows, hardware platform differences, etc. rendering decisions are pushed to the developers by exposing the entire renderer through data files. In other words, shaders, frame layer composition, resource creation, and manipulation are all defined through editable simplified JSON files which are reloadable at runtime allowing for quick iteration cycles and easy experimentation.

Render Configuration

The main entry point to a Stingray renderer is found in files that have an extension of type ".render_config". These render configuration files drive the entire renderer by binding all rendering sub-systems and dictating the composition flow of a frame. They also define quality settings, device capabilities, and default shader libraries to load. Three key ingredients build up these configuration files:

Resource Sets

Resource sets specify GPU resources to be allocated at startup. These are mainly render targets which are aliased by a given name and can be reused throughout the render configuration file.

```
global_resources = [
    { name="output_target" type="alias" aliased_resource="back_buffer" }

    // Regular depth stencil surface
    { name="depth_stencil_buffer" type="render_target" depends_on="output_target" w_scale=1 h_scale=1 format="DEPTH_STENCIL" }
    { name="depth_stencil_buffer_selection" type="render_target" depends_on="output_target" w_scale=1 h_scale=1 format="DEPTH_STENCIL" }
    { name="stable_depth_stencil_buffer" type="render_target" depends_on="output_target" w_scale=1 h_scale=1 format="DEPTH_STENCIL" }

    { type="static_branch" render_settings={ taa_enabled=true } platforms=["win", "ps4", "xb1"]
        pass = [
            { name="stable_depth_stencil_buffer_alias" type="alias" aliased_resource="stable_depth_stencil_buffer" }
        ]
        fail = [
            { name="stable_depth_stencil_buffer_alias" type="alias" aliased_resource="depth_stencil_buffer" }
        ]
    }

    // G-buffer targets
    { name="gbuffer0" type="render_target" depends_on="output_target" w_scale=1 h_scale=1 format="R8G8B8A8" }
```

GLOBAL RESOURCE DECLARATION EXAMPLE

Layer Configurations

Layer configurations dictate the ordering of visible batch submits in the render back-end. They define an array of layers that are processed in the order they are declared. Each layer contains a name used for referencing from the shader system, destination render targets, depth stencil target, and a batch sorting criteria. You can also define a profiling scope for performance analysis of a given layer and interleave resource generator calls throughout the array. In other words, layer configurations define the rendering passes to build up the final frame.

```
layer_configs = {
    default = [
        // Kick resource generator for rendering all shadow maps
        { name="shadow_mapping" resource_generator="shadow_mapping" profiling_scope="shadow mapping" }

        { resource_generator="clustered_shading" profiling_scope="clustered shading" }

        // Clear DST & gbuffer2
        { render_targets=["gbuffer2", "hdr0"] depth_stencil_target="depth_stencil_buffer" clear_flags=["SURFACE", "DEPTH", "STENCIL"] profiling_scope="clears" }
        { render_targets=["hdr1"] clear_flags=["SURFACE"] profiling_scope="clears" }

        // VR depth stencil mask
        { type="static_branch" platforms=["win"] render_settings={ vr_supported=true }
            pass = [
                { resource_generator="vr_mask" profiling_scope="vr_mask" }
            ]
        }

        // Base g-buffer layer, bulk of all materials renders into this
        { name="gbuffer" render_targets=["gbuffer0", "gbuffer1", "gbuffer2", "gbuffer3"] depth_stencil_target="depth_stencil_buffer" sort="FRONT_BACK" profiling_scope="gbuffer" }
        { extension_insertion_point = "gbuffer" }

        // Resolve & linearize depth
        { resource_generator="stabilize_and_linearize_depth" profiling_scope="linearize_depth" }
```

LAYER CONFIGURATION DECLARATION EXAMPLE

Resource Generators

Resource generators represent a minimalistic framework for manipulating GPU resources. They are composed of an array of modifiers executed in the order they are declared. Modifiers can be chained to create advanced effects and are regularly used for post-effects, lighting, shadow rendering, procedural effects, debug rendering, and more. Stingray comes with a toolbox of modifiers such as: full screen pass, compute kernel, mip-map generator, shadow mapping, deferred shading, branching, and so on. Customers with source access can easily create new modifiers.

```
resource_generators = {
    debug_shadows = {
        modifiers = [
            { type="dynamic_branch" render_settings={ shadow_atlas_visualization=true }
                pass = [
                    { type="fullscreen_pass" shader="copy" input=["local_lights_shadow_atlas"] output=["output_target"] dest_rect=[0.0, 0.5, 0.5, 0.5] }
                ]
            }
            { type="dynamic_branch" render_settings={ shadow_cascade_visualization=true }
                pass = [
                    { type="fullscreen_pass" shader="copy:DEPTH_SAMPLER" input=["sun_shadow_map"] output=["output_target"] dest_rect=[0.0, 0.5, 0.5, 0.5] }
                ]
            }
            { type="dynamic_branch" render_settings={ sun_shadow_map_visualization=true }
                pass = [
                    { type="fullscreen_pass" shader="copy" input=["sun_shadow_map"] output=["output_target"] dest_rect=[0.0, 0.5, 0.5, 0.5] }
                ]
            }
        ]
    }
```

RESOURCE GENERATOR DECLARATION EXAMPLE

Stingray currently ships with a high quality deferred renderer with the following frame anatomy:

- Shadow mapping
- G-Buffer population
- Decals
- Reflections
- Lighting opaque surfaces
- Emissive
- Fog
- Skydome merge
- Lighting transparent surfaces
- Post effects
    - Temporal anti-aliasing
    - Depth of field
    - Motion blur
    - Lens effects
    - Bloom
    - Auto exposure
- Scene combine and tonemapping
- Transparency (low dynamic range)
- Present

The render configuration of Stingray's default renderer is found in the Stingray install path under /core/stingray_renderer/renderer.render_config and is completely customizable without the need to recompile the engine code.

VR Support in Stingray

Having a flexible renderer was key to getting initial VR support up and running quickly in Stingray. However, it's important not to underestimate the many challenges in terms of rendering that VR brings to the table.

On mainstream head mounted displays (HMD) such as HTC Vive and Oculus Rift, the requirements are a 90Hz refresh rate with a frame buffer resolution of 2160x1200. To reduce aliasing artifacts the off-screen buffer is in reality super-sampled by a scale of 1.5x in each dimension for a final resolution of 3240x1800. In other words, we only have 11.11ms to shade ~5.8 million pixels for one stereo frame. To put these numbers in perspective it's worth analyzing it by the amount of shaded visible pixels per second:

$$shaded\ visible\ pixels\ per\ second = \frac{(width * height)}{1/Hz}$$

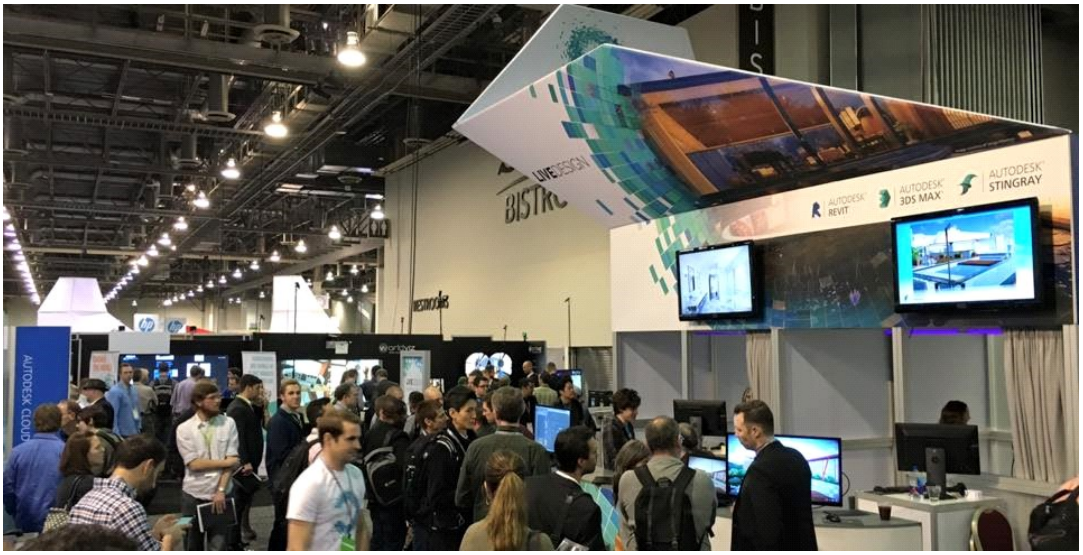
For common resolution types running at 60Hz we observe that VR is more demanding than running a game on a 4K screen at native resolution.

Resolution	Refresh Rate (Hz)	MPixels per second
720p	60	55
1080p	60	124
2160p	60	498
VR (3240x1800)	90	525

Humble beginnings…

Our first implementation of VR in Stingray took the naïve approach of performing sequential stereo rendering. This meant creating two distinct scene cameras to represent both eyes and submitting the entire scene for rendering… twice! It was originally the easiest strategy to follow in order to support and understand different hardware and the particularities of their tracking systems. At this point, new VR software development kits (SDK) were also in beta and quite in flux, so we would usually see considerable updates every 3-4 weeks. Obviously, this wasn't going to be our final iteration on general performance. Due to draw calls and state changes being doubled, this strategy was clearly inefficient. Moreover, this approach was not CPU or GPU cache friendly!

Although we were not satisfied with our initial version, we still demoed and powered the VR experience in the Live Design booth at Autodesk University 2015.



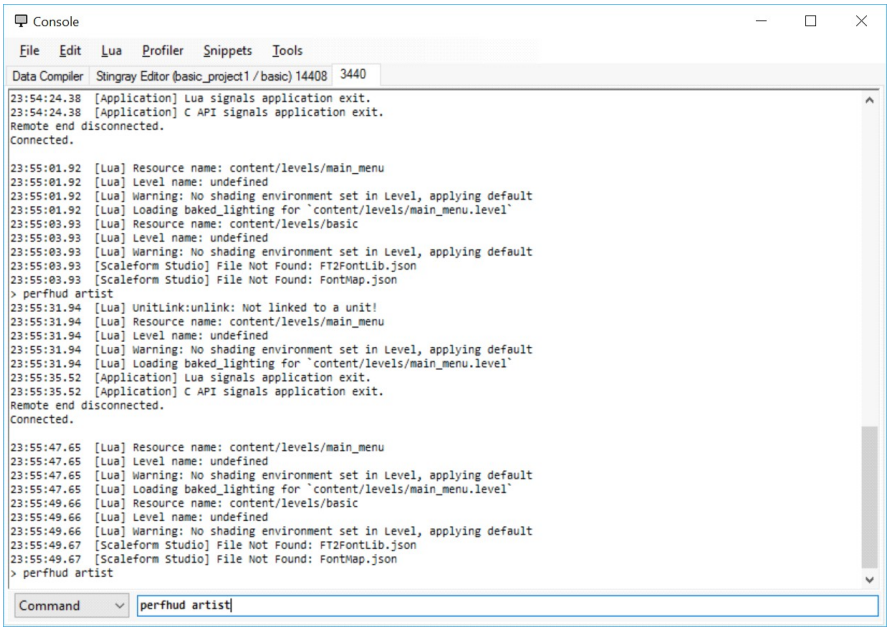
LIVE DESIGN BOOTH AUTODESK UNIVERSITY 2015

http://www.cgarchitect.com/2016/02/2016-the-year-of-vr-stingray-and-live-design

Stingray Profiler

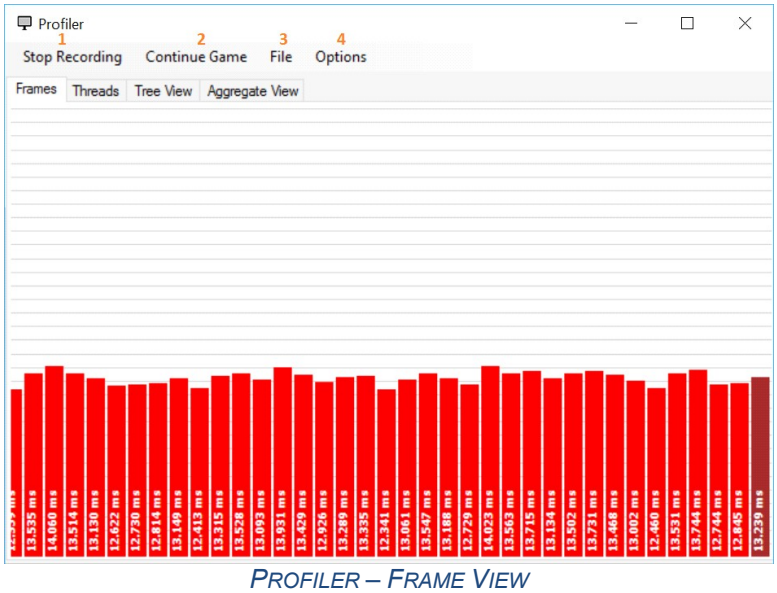
Before we delve into VR optimizations developed over this past year, let's get familiar with the existing performance tools that are available in Stingray today.

The Profiler is accessible through the external console (by pressing Alt+2) and should be your first stop when analyzing your scene. It gives you a frame breakdown of all the work the engine is doing on each thread, including the GPU. Note however that to enable the GPU profiling trace of your application, you need to activate the artist performance heads up display (HUD). To do this, type "perfhud artist" in the command line found near the bottom of the external window.



EXTERNAL CONSOLE

The Profiler window lets you stop recording profiling data (1) or pause the running instance of the engine (2) for trace inspection. It also permits you to save and load profiling data (3) in order to share these traces back and forth between people, as well as to set some settings (4) such as breaking when a frame is longer than a certain value.



PROFILER – FRAME VIEW

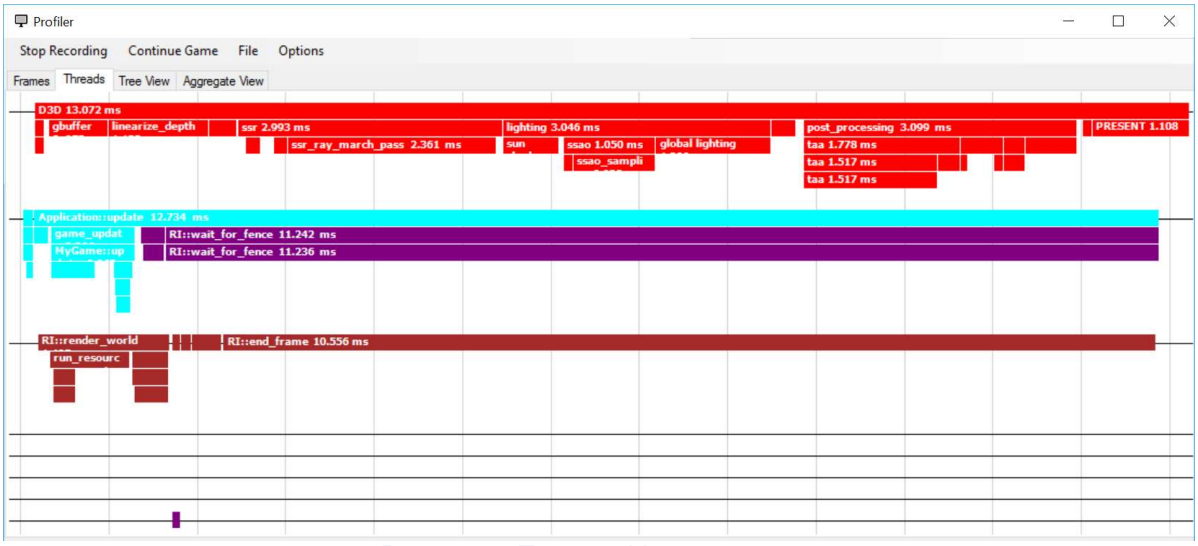
The Profiler is divided into four different views:

Frames View

This view displays the total time for each rendered frame, letting you easily detect spikes in specific frames that are usually worth inspecting.

Threads View

This view presents all threads available on your system and shows the scoped timings recorded for specific work that the current frame has done. If you're currently capturing frames, the Threads view always displays the latest frame rendered. By pausing the capture, you can navigate in history using the left and right arrow keys or by using the frame view.



PROFILER – THREADS VIEW

Assuming you have the artist performance HUD running, the top three threads are always the GPU (red), followed by the main game (cyan), and rendering (brown) threads. Below are all other threads available on your system, either at work or available to take incoming jobs spawned by the main or render threads. With very small scopes, you might have to zoom in to see more details of the work being performed. Next to the name of each scope, the amount of time to perform this particular job displays in milliseconds.

Tree View

The Tree view gives you the same information as the Threads view, but it's displayed as a tree showing the timing in milliseconds at the right of each individual scope.

PROFILER – TREE VIEW

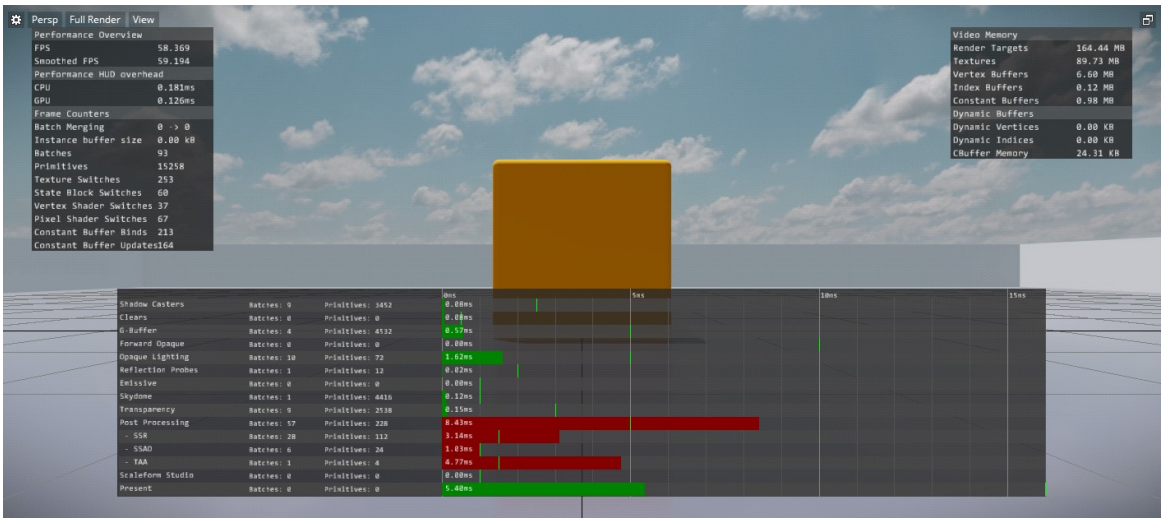
Aggregate View

The Aggregate view is similar to the tree view, but it gathers and computes scope timing over several frames using a specific formula. This view is useful to find either the maximum, minimum, or average times for the scopes shown.

PROFILER – AGGREGATE VIEW

Artist Performance HUD

The Artist Performance HUD debug mode (also accessible using the perfhud artist command) gives a general overview of your application's performance. It displays information such as the frames per second, batch merging, total batch count, number of rendered primitives, total shader complexity information, and video memory usage. This can help you quickly understand the problematic elements of your scene.



ARTIST PERFORMANCE HUD

Going back to an initial profiling trace when performing sequential stereo rendering, this would be our typical frame breakdown:

SEQUENTIAL STEREO RENDERING PROFILING TRACE

As observed above, the amount of work is clearly doubled on both the GPU and render threads.

VR Optimizations

To improve general VR performance, we implemented three major improvements this past year that are now part of the latest version of Stingray.

Compound Frustum

The camera frustum is used during different stages of our pipeline. First, we use it to optimize our rendering by performing frustum culling. This step gathers and prepares all objects for rendering that are either partially or totally inside the camera view volume while the others are discarded. We then use the frustum to compute cascaded shadow maps in order to reduce perspective aliasing. In terms of work, this means we slice the frustum into four different parts and compute a new shadow map for each different section of the view volume. Finally, we use the frustum during our clustered deferred shading pass. This step voxelizes the view volume and stores the different local light

contributions inside the affected voxel buckets. This data structure is then passed to the shaders to compute the final local light contribution. Doing these three different actions twice for each eye is costly and unnecessary. Considering that the general view direction of each eye is equivalent and the interpupillary distance is quite small, it's worth computing a single enclosing frustum to minimize computations on the render and GPU threads.

Hidden Area Mask

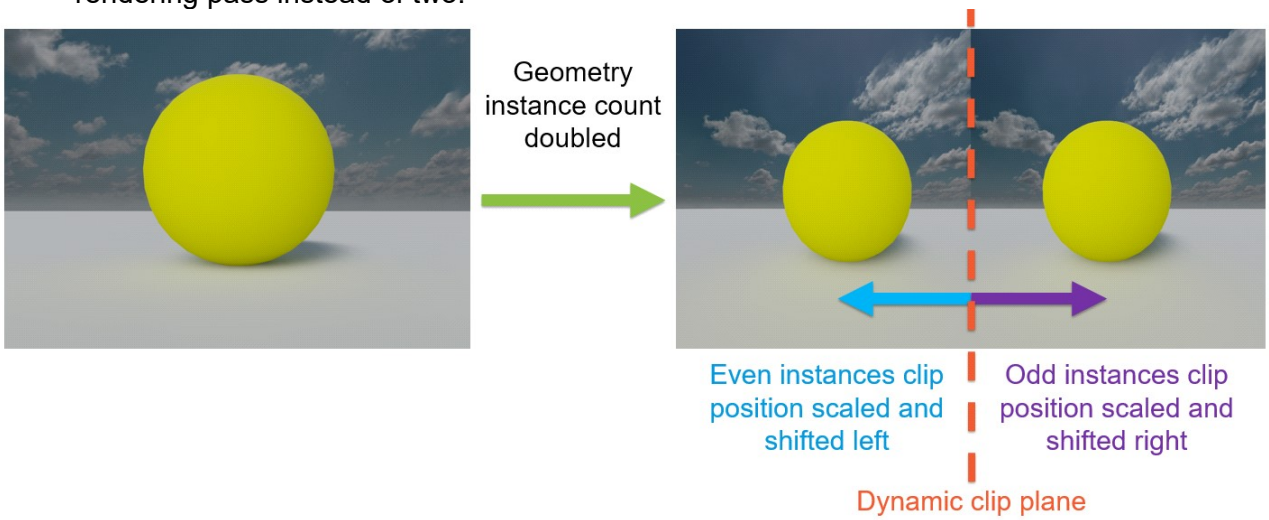
The Hidden Area Mask uses a mesh to early-out on pixels that aren't visible in the final image as seen through an HMD. In other words, this is the first mesh to be drawn in our depth stencil buffer when in VR mode. This ensures that we cull out geometry that you can't see and applies post processing only on visible pixels. The Hidden Area Mesh is provided through the HMD software development kit and is only available for the HTC Vive.

BLACK PIXELS REPRESENT THE HIDDEN AREAS ON THE HMD

Instanced Stereo Rendering

Instanced Stereo Rendering is an optimization that makes it more efficient for Stingray to render stereoscopic images for VR headsets. It uses hardware accelerated geometry

instancing to produce both the left and right eye version of the scene during a single rendering pass instead of two.



INSTANCED STEREO RENDERING PIPELINE

Distinct eye transforms are stored in the constant buffer and stereo rendering is handled by the vertex shaders. Even instances use the left eye matrices and are shifted left, while odd instances use the right eye matrices and shifted right. The clip plane is dynamically adjusted to prevent spill over to opposite eyes. Instanced stereo rendering is now built in to the default renderer provided with Stingray. All material and post effect shaders have been VR-enabled to implement this optimization.

Generating the VR profiling trace of the previous scene with all these VR optimizations enabled gives:

OPTIMIZED STEREO RENDERING PROFILING TRACE

Comparing the optimized results to the previous sequential stereo rendering trace, we observe gains close to 35% on the GPU thread and 45% on the render thread.

Building VR Experiences using Stingray

The Stingray engine has all of the VR systems implemented in it for the Oculus Rift, the HTC Vive, and other mobile VR devices. Using the Stingray Editor, you can use these systems to build truly immersive experiences for VR in a simple and intuitive fashion. In this section, we'll describe how the VR templates can get you started in building your own projects and also show a few examples of how to create functionality for VR using the Flow visual scripting language.

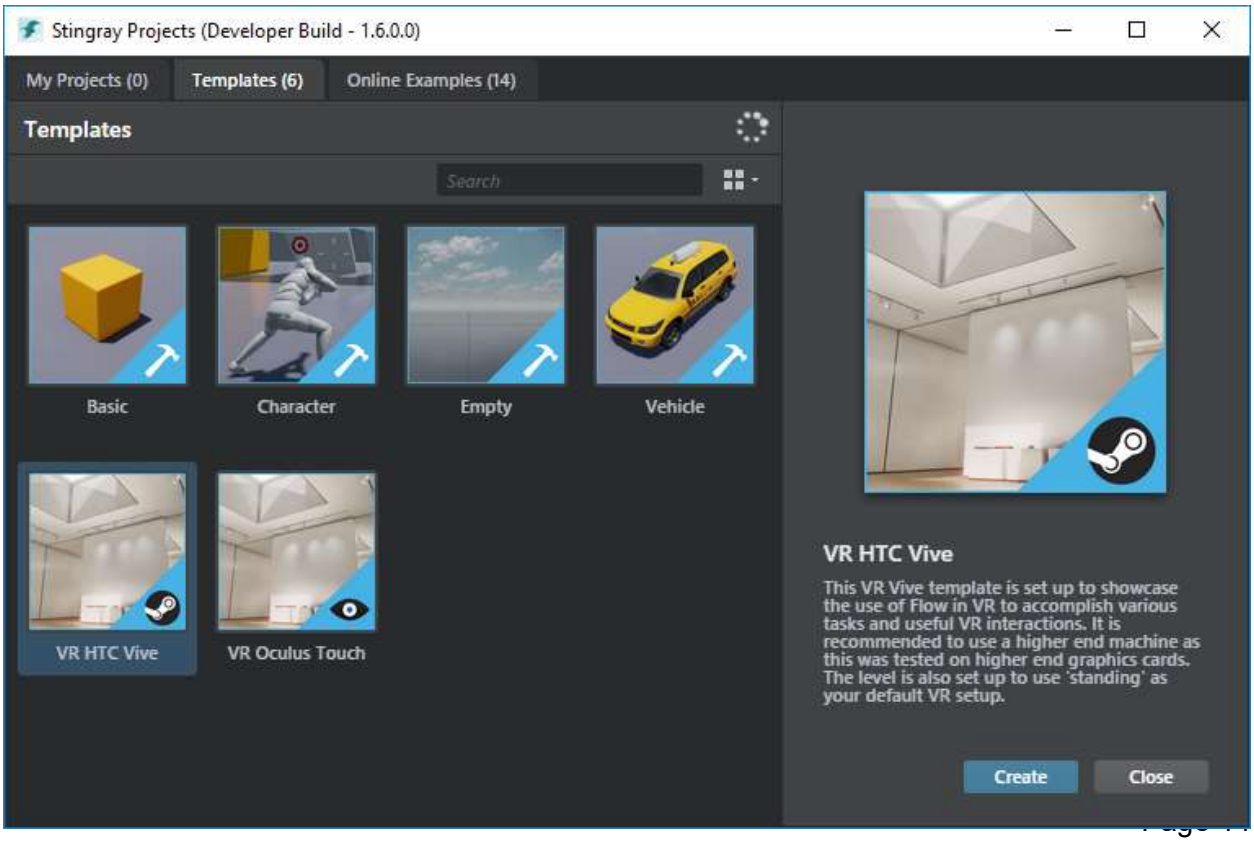
AUTODESK® STINGRAY

Stingray VR Templates

The Stingray Editor ships with a few sample projects to help you get started, including templates for both the HTC Vive and Oculus Rift. In general, the best way to start a new VR project is always to clone one of these VR templates rather than starting completely from scratch, as there is a lot of things set up in the templates that you don't want to worry about. By default, template projects install in the following location:

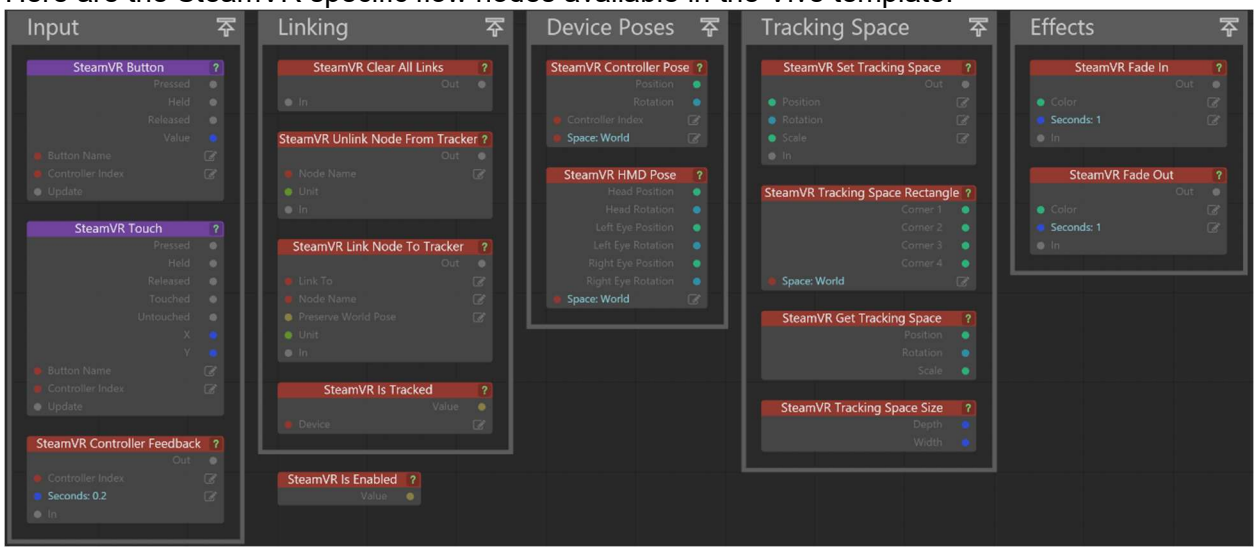
```
C:\Program Files\Autodesk\Stingray\<version>\editor\templates
```

They're also available under the Templates tab in the Project Manager every time you start Stingray.



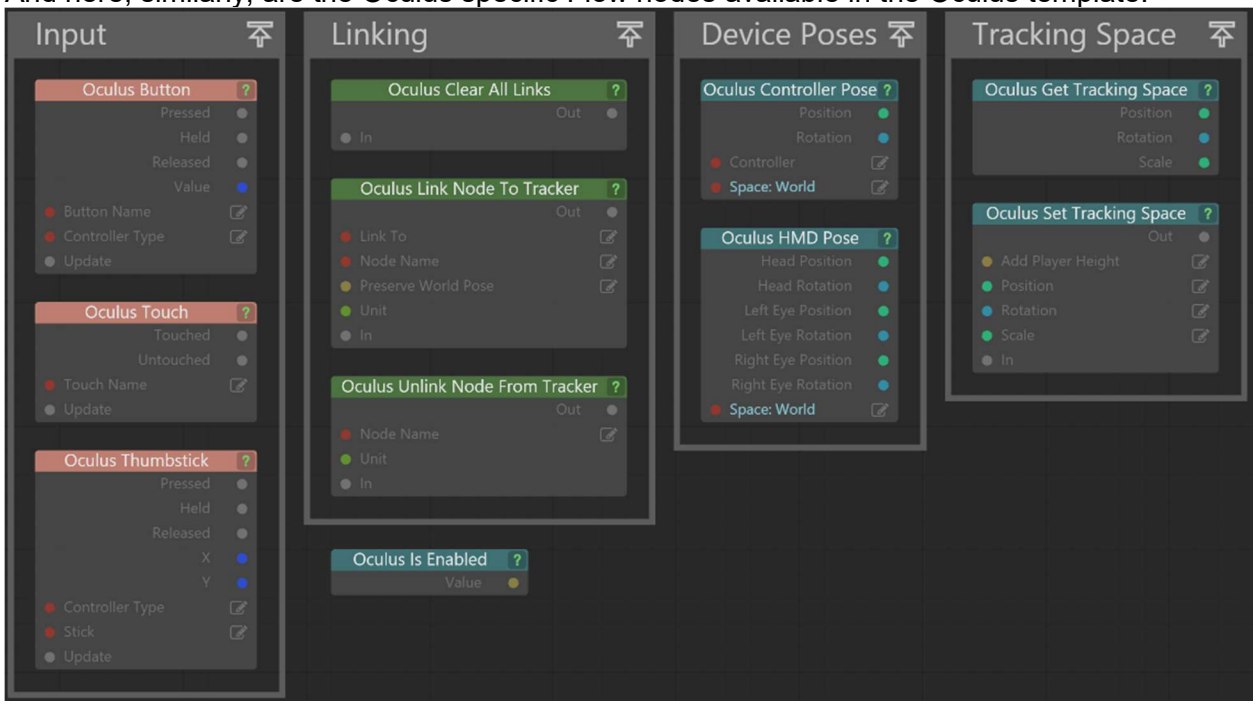
Although there are separate project templates for both the Vive and Oculus, we kept them as similar as possible to make it easy to develop for one or the other. In Stingray, you can build your desired behaviors and mechanisms using either Lua (if you are experienced with programming), or the visual node-based Flow scripting language. The Oculus and Vive have an almost one-to-one Lua API, as well as a very similar set of flow nodes.

Here are the SteamVR specific flow nodes available in the Vive template:



STEAMVR FLOW NODES

And here, similarly, are the Oculus specific Flow nodes available in the Oculus template:



OCULUS FLOW NODES

Although there are a few discrepancies between the Oculus nodes and those of the Vive, most of the essential functionalities remain the same. These nodes, in conjunction with the other default Flow nodes, cover all of the functionality necessary to implement any behavior you want for either device.

- The “Input” nodes deal with any button presses, touch input, or haptic feedback.
- The “Linking” nodes deal with attaching objects in your scene to the devices that are tracked by the VR system, such as your controllers or Head Mounted Display (HMD).
- The “Device Poses” nodes get you information about where your tracked devices are located and how they are oriented.
- The “Tracking Space” nodes deal with mapping the real-world tracking space defined by your VR system into the virtual environment of your project.
- Effect nodes for the Vive let you do simple Fade In/Fade Out effects.

VR Initialization in Lua:
Aside from the Flow nodes defined in the different VR projects, there are also the *steam_vr.lua* and *oculus.lua* Lua files in which the respective VR systems are initialized in the engine and the appropriate cameras needed for rendering are created. These files are necessary for VR to work.

```
function SteamVRSystem:initialize(near, far, hmd_rt, hud_rt)
    if not SteamVR then return end
    if not self then return end

    -- Create main vr camera
    self._vr_main_camera_unit = World.spawn_unit(self._world,
    self._vr_main_camera = Unit.camera(self._vr_main_camera_un
    Camera.set_near_range(self._vr_main_camera, near)
    Camera.set_far_range(self._vr_main_camera, far)

    -- Initialize VR system
    self._is_initialized = SteamVR.initialize(hmd_rt, self._vr

    -- Bail if there was an error on initialization
    if not self._is_initialized then
        World.destroy_unit(self._world, self._vr_main_camera_u
```

```
// Note: Adjust render settings below
render_settings = {
        // Set renderer to support VR
        vr_supported = true
        // Specify HMD resolution
        vr_hmd_resolution = [ 2160, 1200 ]
        // Scale of VR render target for
        // super sampling to reduce aliasing
        // artifacts
        vr_target_scale = 1.6
        // Control mirror window mode: mono
        // or stereo
        vr_mirror_mode = "mono"
    }
}
```

Settings.ini and VR settings:
Each Stingray project also has a *settings.ini* file at its root, and in the VR templates it’s used for things like telling the renderer we want to actually render for VR, specifying the render target scale for supersampling purposes, or even how the mirror window (which is the desktop window of what you see in VR) should behave.

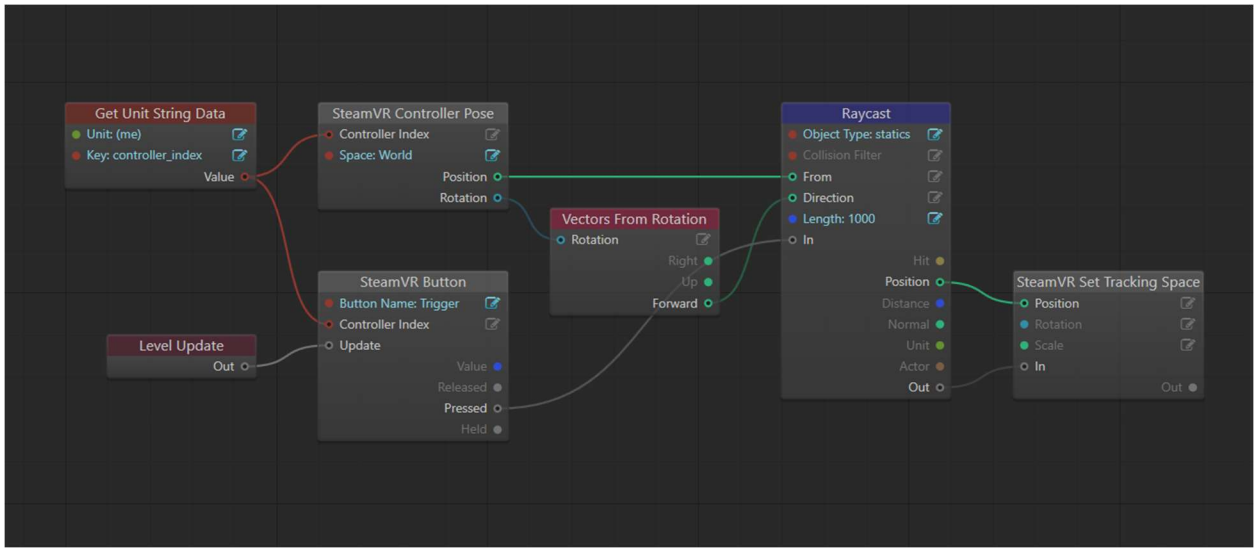
Finally, the VR templates also come with a lot of premade functionality built in Flow for doing things like picking up objects, teleporting, playing sound effects, and setting up the controller units. These Flow graphs are encapsulated in their own Unit objects, but are still quite complex. Instead of trying to explain how the entirety of the Flow found in the VR templates works, we can break it down into a few simplified examples to give you an idea of how the overall project works.

Note: Examples are done using the SteamVR template, however they can be done as simply in the Oculus template using those nodes instead.

Flow Example: Teleportation

As a first example, we'll take a look at how to create a simple teleportation mechanism. In VR, people can start feeling nauseous if there is mismatched motion between what they see and their actual movements. Moving around the level with thumbstick input like a traditional video game, for example, doesn't lead to good experiences in VR. Teleportation is a good way to let people move around a level without introducing that feeling of nausea.

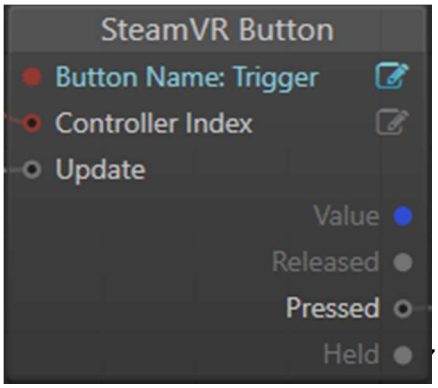
The basic functionality for our teleport mechanism is as follows: When the user presses a button (for example, pulls the trigger), they teleport to the location they are pointing to with the controller. Simple enough. Let's take a look at the flow graph for that:

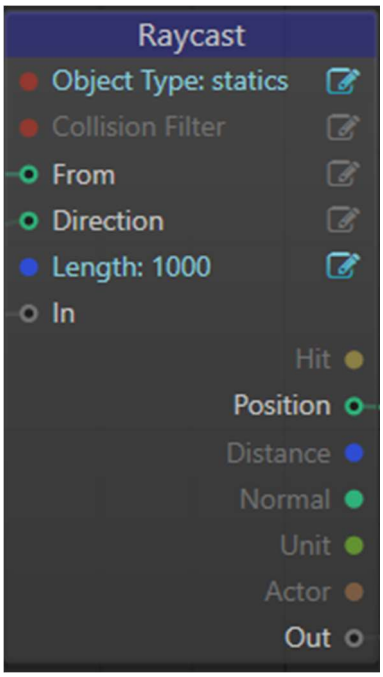


FLOW GRAPH FOR A SIMPLE TELEPORTATION MECHANISM

Let's break down what's happening:

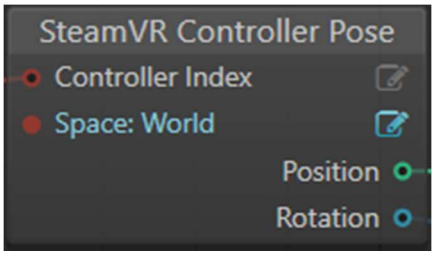
First, we detect when input is pressed by using the **SteamVR Button** node. This node detects when a specified button is pressed, held, or released, as well as the value of the pressed button (0 for unpressed, 1 for pressed, or in between for analog buttons such as the trigger). In our case, we want to detect when the trigger is pressed and lead into the **Raycast** node.





We use the **Raycast** node to get the location of where the user is pointing the controller. It shoots a ray from a specified position in a specified direction until it hits something or exceeds the maximum length we set.

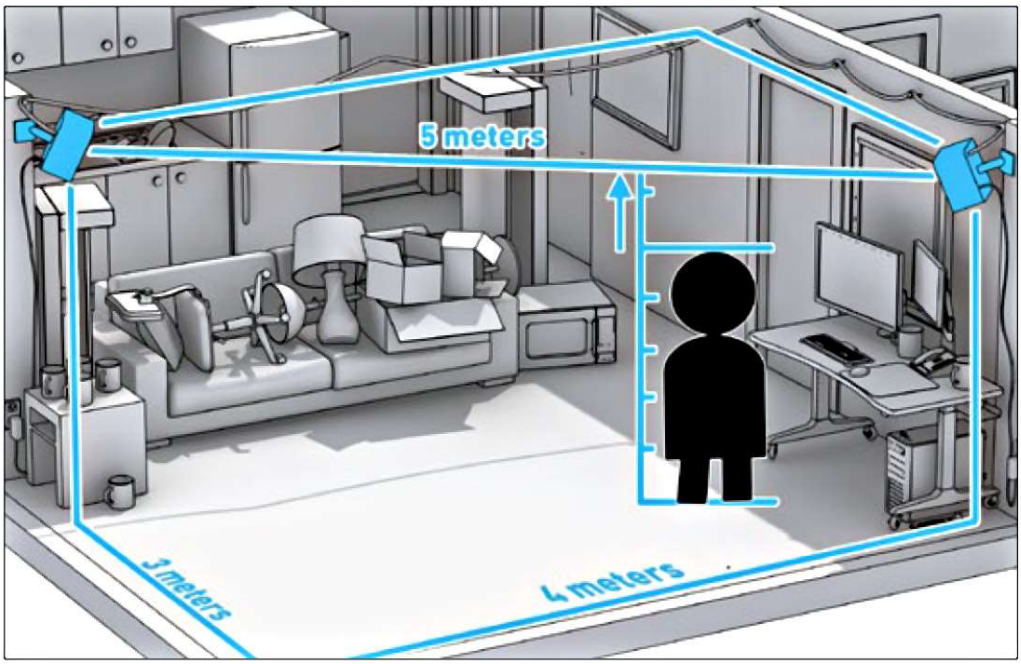
In our case, we want to shoot the ray from the controller's position, in the direction that controller is facing. We can get exactly that information with the **SteamVR Controller Pose** node.



We use the controller position as the start position for the raycast, and the forward vector of the controller's rotation for the direction of the raycast.

The ray is shot using those parameters, and returns a lot of information about the hit location.

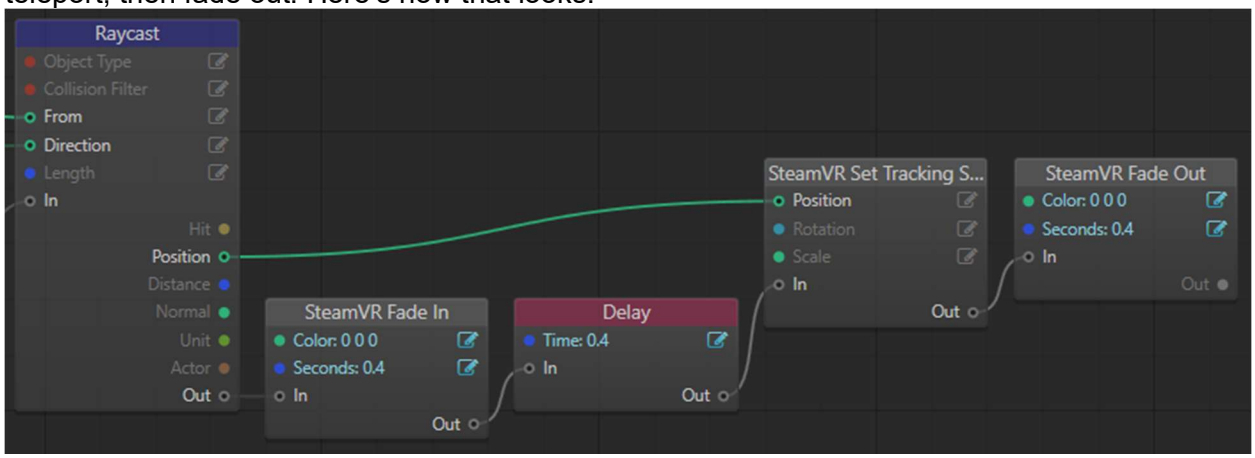
We can then use the raycast hit position to set the new SteamVR tracking space position. With the Vive, you can define the play area that people can move around in. The lighthouse sensors that come with the Vive create a bounding volume in which you can track the HMD and controllers. When we set the tracking space position, we are setting where this play area maps to in our virtual environment.



As you can see, it's quite simple to get a very basic teleport up and running. It's far from perfect though, and there's a lot of room for improvement. Let's look at a few ways we can make it better.

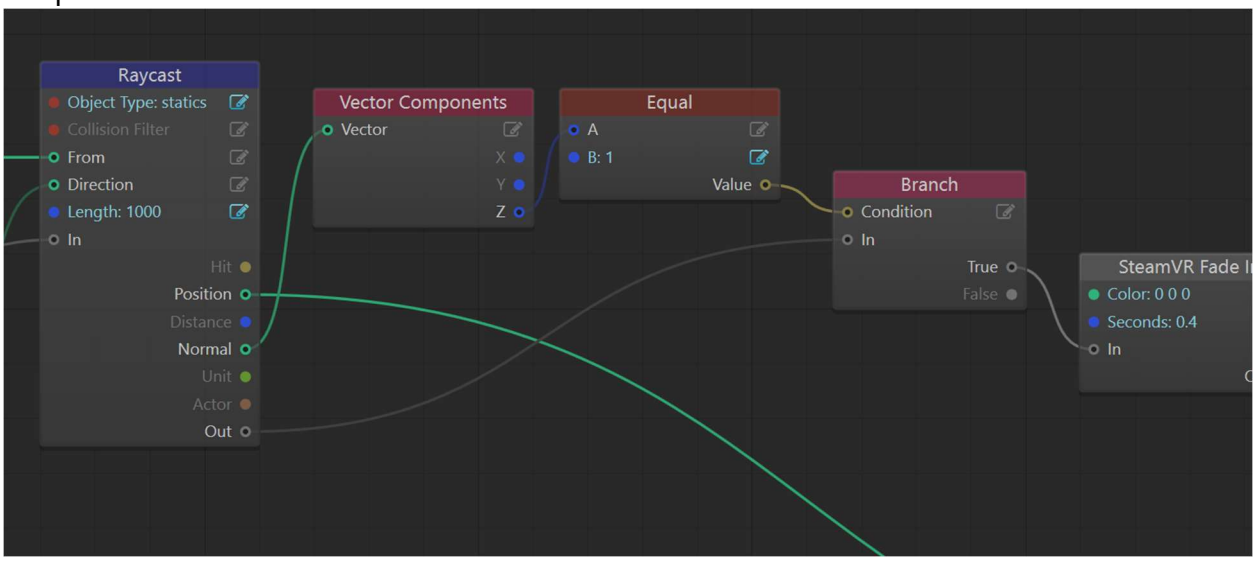
Fade To Black

The teleport works fine using the Flow we went over above, however such an abrupt jump from one location to another can disorient users. In order to ease the transition, we can use the **SteamVR Fade In/Fade Out** nodes to fade the screen to black, perform the teleport, then fade out. Here's how that looks:



Flat Surfaces

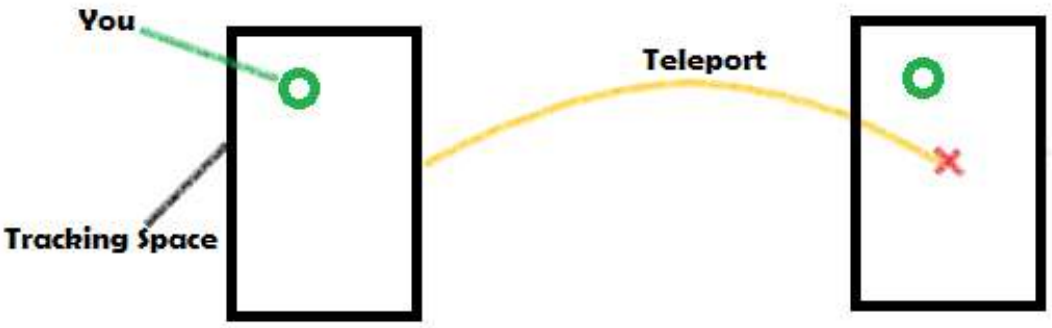
Another improvement we can easily add is to restrict teleportation to flat surfaces facing upwards. That way, people can't teleport when they're pointing at a wall or ceiling. Doing this is trivial with the Raycast node. We get the information about the surface normal at the raycast hit position, check if the normal is facing up, and if it is continue with the teleportation. Here's how that looks:



Local Position Adjustment

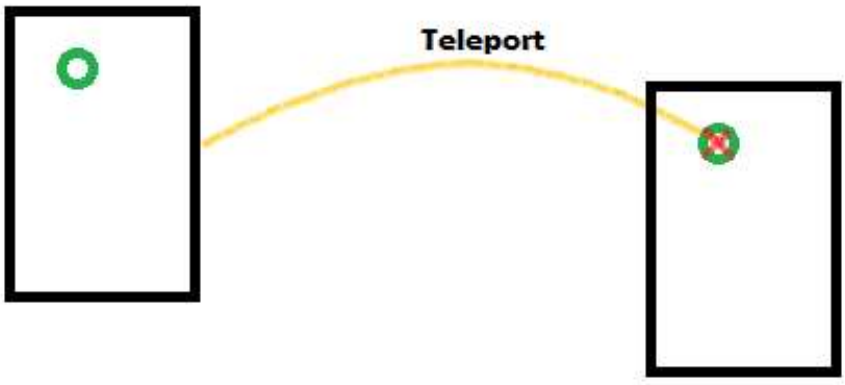
One last small improvement, which is actually quite important, is to offset the teleport destination by our HMD's local position within the play area. It may not be immediately clear why we'd want to do that, so here are some diagrams to explain.

This is how we teleport **without** the local position adjustment



The way we have the teleport mechanism set up now, we point to a location (the red X) and set our tracking space to that location. However, the user (represented as a green circle) can be anywhere within that tracking space. As you can see, when we set the tracking space to the teleport destination, the user doesn't end up where they were pointing.

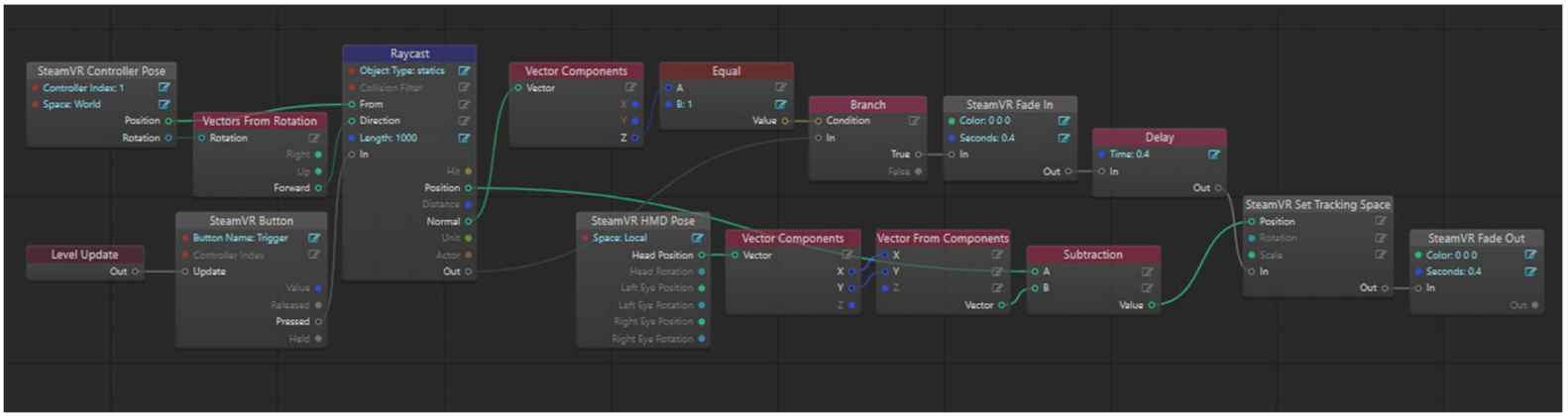
This is the teleport **with** local position adjustment:



By shifting the position, we set the tracking space to be the local position of the HMD in the tracking space, our teleport destination is exactly where the player ends up, and the player gets the experience they expect.

Final Flow Graph

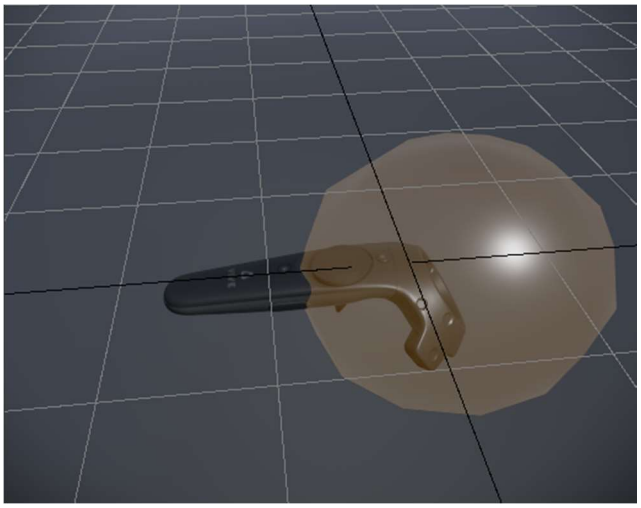
This is the final result of all of the improvements put together:



There are still other improvements we could add (such as visualization of our teleport destination), but this is a solid teleport functionality nonetheless, and as we've seen, nothing too complicated.

Flow Example: Picking Up Objects

For our second example, we'll look at how we can pick up and interact with dynamic objects in our scene. To start off, we'll need a way to know when the controller is close enough to an object to be able to pick it up.

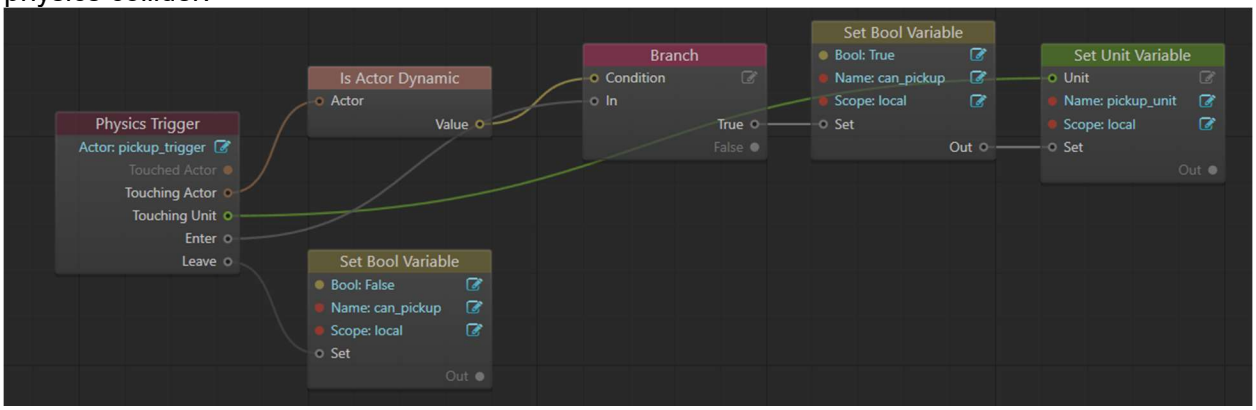


PHYSICS TRIGGER ON THE CONTROLLER UNIT VIEWED IN THE UNIT EDITOR

As you can see here, we added a sphere mesh to the controller. (Normally this mesh is invisible, but made visible for demonstration purposes.)

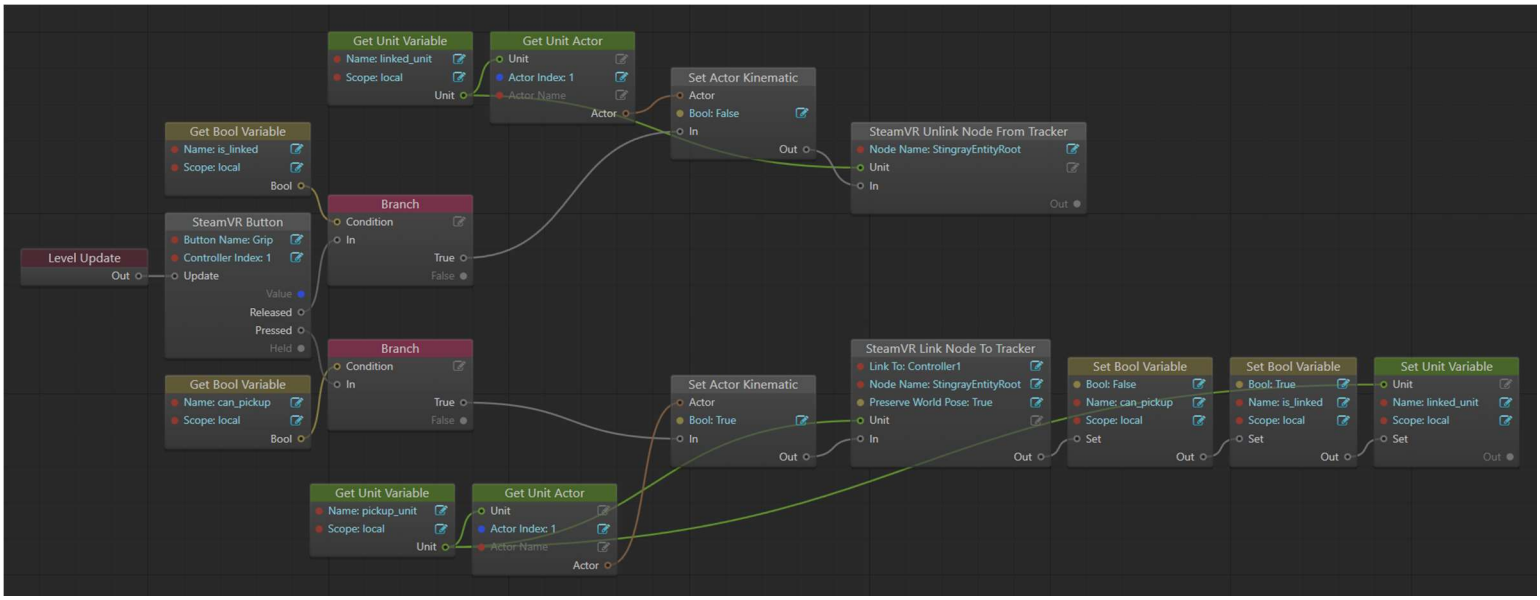
This sphere mesh has a physics actor associated with it, which we can use as a physics trigger. Whenever an object enters this sphere, it triggers an event in Flow. We can then store necessary information about this nearby object so that we'll be able to pick it up on button press.

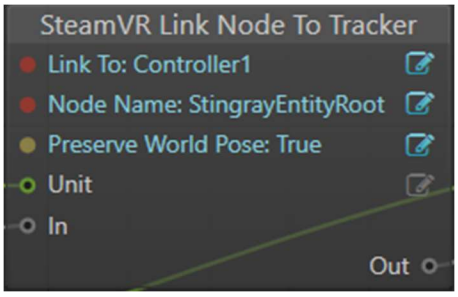
This is the Flow graph of the trigger that happens when an object intersects or enters our physics collider:



When an object collides with our trigger, we set a flag stating we can currently pick up an object, and store the object we want to pick up. We also verify that the object is dynamic. We don't want to pick up static objects, otherwise we'd be able to pick up things like the floor (which, hey, could be a feature!)

The next step is to actually pick up the object we're touching when we press a button. This is how the Flow looks for that:





It seems like a lot is going on here, but most of it is handling input, making sure our flag that defines if we're touching something is set, and getting the actual object that we're touching. The important part of this graph that does most of the heavy lifting is the **SteamVR Link Node To Tracker** node (and **SteamVR Unlink Node From Tracker** when we let go).

Now the naïve way to accomplish having an object follow your controller is to simply set the position of that object to the position of the controller. However, due to the fact that we have a multithreaded game engine running our game, it's not that simple.

In our engine, all of the game logic and calculations are done on one thread, and all of the rendering happens on another. In order to ensure that we never starve the GPU and that we always have resources ready for it to consume, the render thread actually deals with data that is 2 frames behind the game thread. If the render thread dealt with data processed for the same frame as the game thread, we run the risk of the GPU waiting for the CPU to finish its calculations.

In our VR systems, all of the tracked positions are updated on the render thread to make sure that we always see tracked objects exactly at the position that the Vive or Oculus systems have them tracked at (including all of the position predicting that those systems do). So, if you set an object to the controller's position in Flow or Lua (which happens on the game thread), that object will have a slight lag as to where it should actually be. Using the **SteamVR Link Node To Tracker** node specifies that this object, which you are linking to a tracked device, should have its position updated on the render thread. To demonstrate the difference, here's an object that follows the controller using two different methods:

- The red ball follows the controller by setting its position on the game thread
- The blue ball follows the controller using the **SteamVR Link Node To Tracker** node

In this image, the controller is still. Both the red ball and the blue ball are on top of each other at the correct position.

Once we start moving the controller to the left, we can see that the blue ball stays appropriately locked at the position it should be relative to the controller, whereas the red ball drifts slightly behind.

To summarize, picking up objects is as simple as using the **SteamVR Link Node To Tracker** Flow node! There is some set up involving physics triggers and knowing when we can actually pick something up, but once that's done you are ready to go.

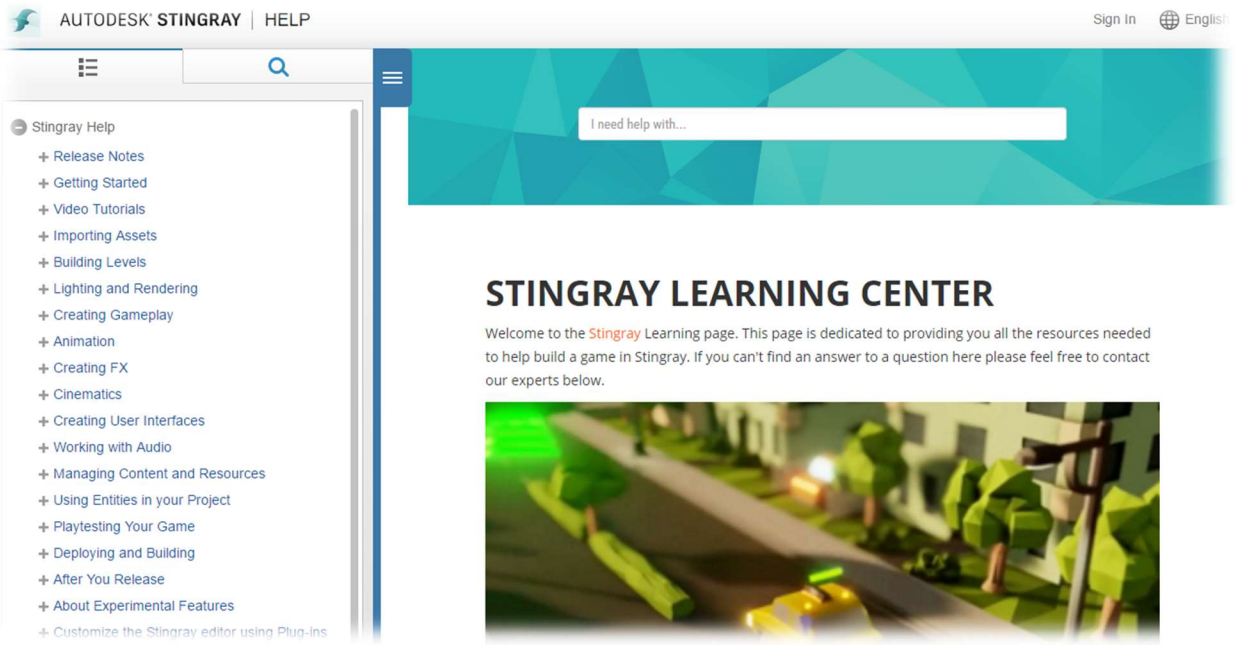
Other Ideas

So that was a pretty in-depth look at two different functionalities built with Flow, but hopefully you see that it's not too complicated to get things working how you want them. Using what we covered in this section, you can implement many other really useful features.

One such example would be a gaze mechanism. Using the Raycast node and feeding in the HMD position and its forward vector, you can find out what object you are looking at and make a descriptive label appear on top of it, for example. Or, if you wanted to improve the teleportation mechanism, you could make a unit appear at the raycast hit position in order to give visual feedback to the user about where they are about to teleport.

Documentation and Online Resources

With Flow, you can create in-depth functionality without ever needing to touch code. The only thing you need to get started is a basic understanding of what the different nodes do, and the online resources and documentation can help you get on your way in no time. There's documentation for the Lua API and all the different Flow nodes as well as various different tutorials to help you get started creating your own projects!



Visit the Stingray Learning Center here: **http://help.autodesk.com/view/Stingray/ENU/** or access it from the "Help" menu in Stingray.

Optimizing VR Content

Maintaining high framerates is paramount in VR. Unlike traditional interactive experiences, good performance in VR is not only desired, it's mandatory. Poor performance will make your users sick.

Next, we'll go over tools and procedures to help you take a poorly performing VR project and make it work. Let's say you've been given scene, such as an arch-viz interior originally built for offline rendering, and you've been tasked with making it run in VR.

Optimization Checklist

We'll start with this checklist of typical problems:

- Is overdraw set too high?
- Am I running too many expensive post-processes?
- Am I rendering too many polygons?
- Am I running out of GPU texture memory?
- Am I casting too many shadows? (batches)

In general it's a good idea to reduce or eliminate all non-essential rendering costs in order to get good baseline performance. Once everything renders at a consistent framerate for VR, start adding optional features one by one, making sure each additional feature does not adversely impact performance.

Render Settings – overdraw and anti-aliasing

Anti-Aliasing and overdraw settings are stored in your settings.ini file, under render_settings.

```
// Note: Adjust render settings below
render_settings = {
  // Set renderer to support VR
  vr_supported = true
  // Specify HMD resolution
  vr_hmd_resolution = [ 2160, 1200 ]
  // Scale of VR render target for super sampling to reduce aliasing artifacts
  vr_target_scale = 1.4
  // Control mirror window mode: mono or stereo
  vr_mirror_mode = "mono"
 }
```

You can choose to set taa_enabled or fxaa_enabled. Start with FXAA, which is much less expensive. By default, Stingray templates have TAA enabled.

The other setting is overdraw. Overdraw is a multiplier on your output resolution. If overdraw is set to 1.5, the image is rendered 1.5 times the size of the output resolution, then downsized as an anti-aliasing method.

Stingray VR templates run at 1.6, but you can lower this to at least 1.4 as a starting point.

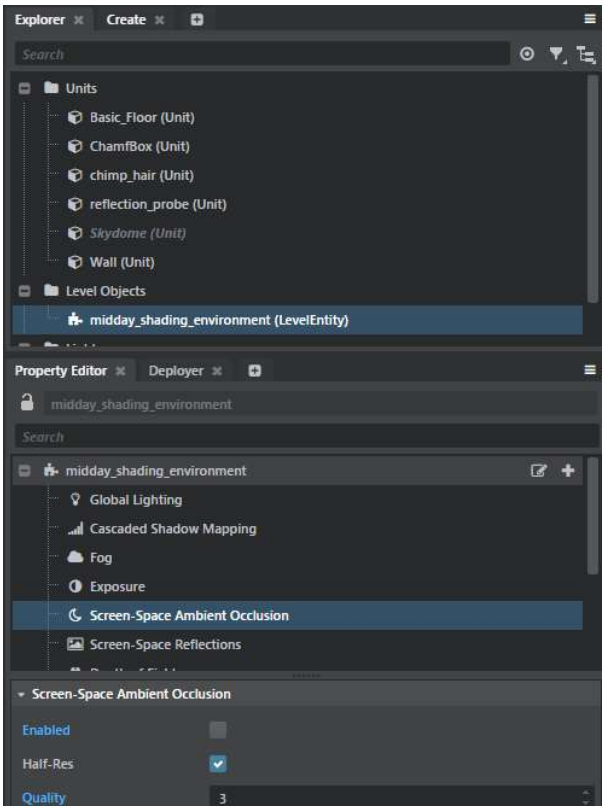
Post Processes

All post-processes come at a cost. Some of them are prohibitive to all but the simplest of VR projects:

- Temporal Anti-Aliasing
- Screen Space Ambient Occlusion
- Screen Space Reflections
- Bloom
- etc

When optimizing your VR project, first turn off all post-processes until all areas of your scene are performant, and you have milliseconds to spare for enhancing the quality of your scene with post processes. At that point, you can start turning on post-processes one at a time, making sure that you aren't losing necessary framerates.

Most post processes can be disabled by selecting the shading environment entity in your level and disabling each post process in the Property Editor.



The Stingray VR templates have only Bloom and Auto-Exposure enabled by default.

Poly Reduction

Typically polygon count is not that much of problem, in fact it's usually the last problem. Stingray can handle a lot of polygons very efficiently.

Arch-viz interiors usually have very reasonable poly counts, aside from specific assets. Assets from photogrammetry for example, or plants, and sometimes insignificant models like chain links that happen to have tons of polys can simply be replaced.

But, if you do need to do some poly reduction, here are some Autodesk tools you can use:

Tool: 3ds Max

- Optimize
- ProOptimize

Tool: Maya

- Polyreduce

As well as some very good commercial grade poly reduction tools:

Tool: SimplyGon

- https://www.simplygon.com/

Tool: Instant Field Aligned Meshes

- https://github.com/wjakob/instant-meshes

Texture Blowout

When you've used up all your GPU texture memory, you'll have to swap to CPU memory. Swapping is very slow and severely impacts performance.

You'll run into this a lot with content that was originally made for offline rendering, where using up GPU memory is not an issue. If you're used to real-time rendering, some of these textures can seem shockingly large and wasteful.

One reason that you will run into this issue is that all textures when imported into Stingray get converted to DDS. DDS is the native file format for GPUs, and by default they are uncompressed. So textures that are reasonable for offline rendering become unreasonable for real-time rendering very quickly.

The first thing you'll need to do is determine if you are in fact running out of texture memory. GPUs typically come in sizes like: 2GB, 4GB, 8GB, and so on.

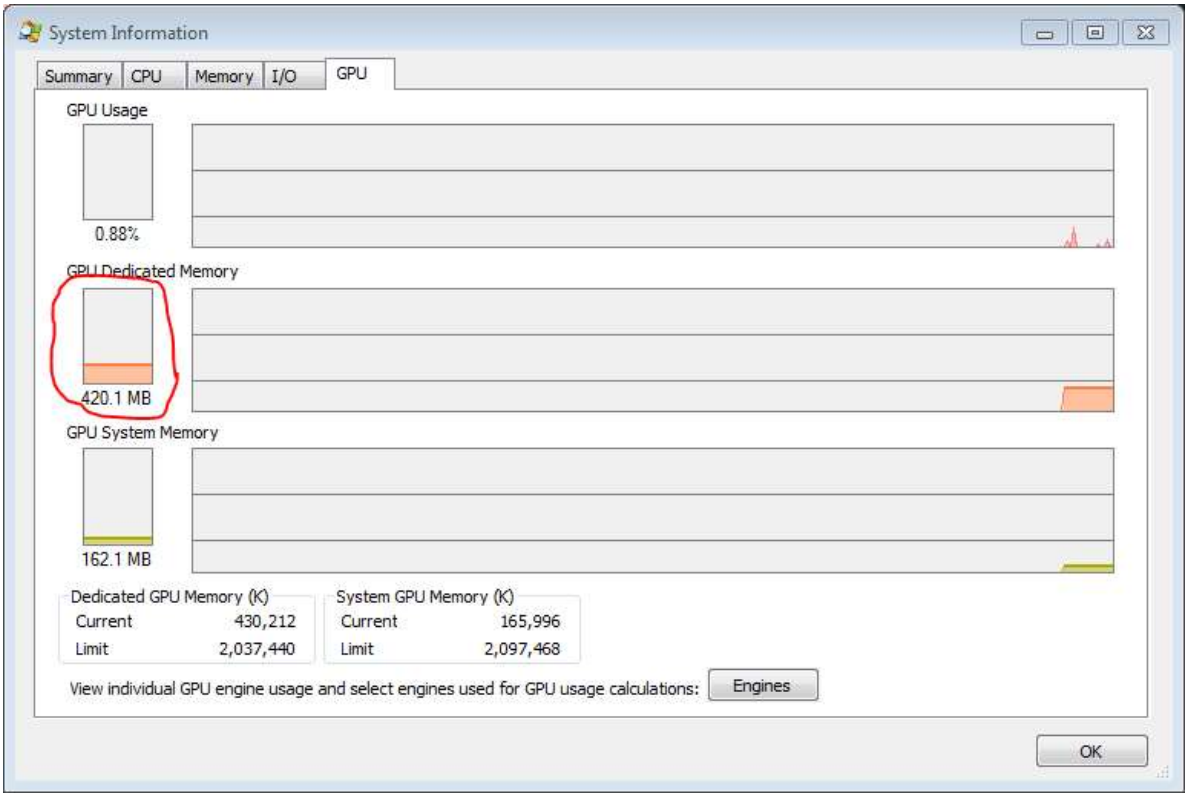
You can use a tool like Process Explorer to quickly see if you are running out of texture memory. This is like the Windows Task Manager, but much more detailed.

Tool: ProcExp

Available on Microsoft technet:

- https://technet.microsoft.com/en-us/sysinternals/processexplorer.aspx

Open Process Explorer and click on the GPU Tab to see a 'GPU Dedicated Memory' bar. If that bar is full, then you have exceeded your texture memory.



PROCESS EXPLORER GPU TAB

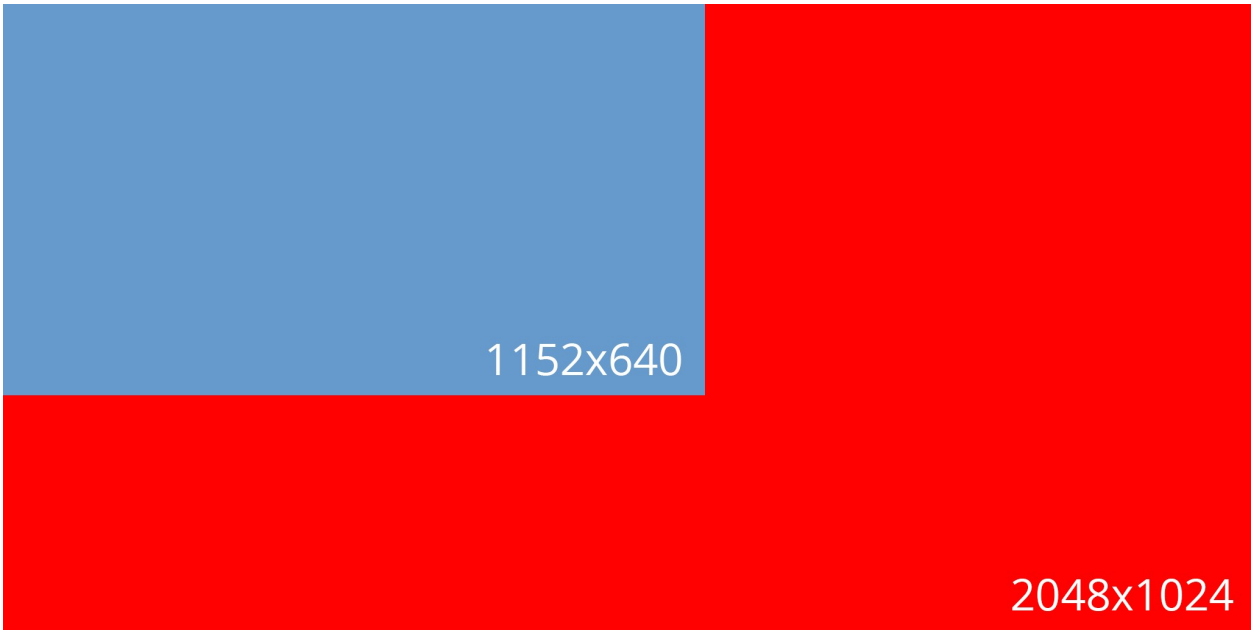
Texture resizing

Once you've confirmed that you have a problem with texture memory, you'll want to first resize your textures.

Most compression algorithms require texture dimensions to be either a multiple of four or a power of two.

Resize all your textures to a power of two: 64x64, 128x128, 1024x512, and so on.

Stingray resizes textures (up to the next highest) to meet compression requirements, but this wastes memory without improving quality.



ORIGINAL TEXTURE (BLUE) IS RESIZED TO THE NEXT POWER OF TWO (RED)

Tool: XnView

This is a Windows tool for viewing and processing images that shows all textures in a project recursively. You can sort by size or type, and batch process images quickly to resize many textures at once.

Show Files in Subfolders

To show all files in subfolders, click 'View' and choose 'Show files in subfolders'.

You may also need to enable the filter for images only. In the browser panel, click on the filter dropdown and choose images.

Batch Process Resizing

To batch process images, click 'Tools' and choose 'Batch Processing'.

To resize images, click on the Transformations tab, and choose Resize under the Image section. Click 'Add' to add it to the process step, choose the desired width and height, and press 'Go' to process.

Texture Compression

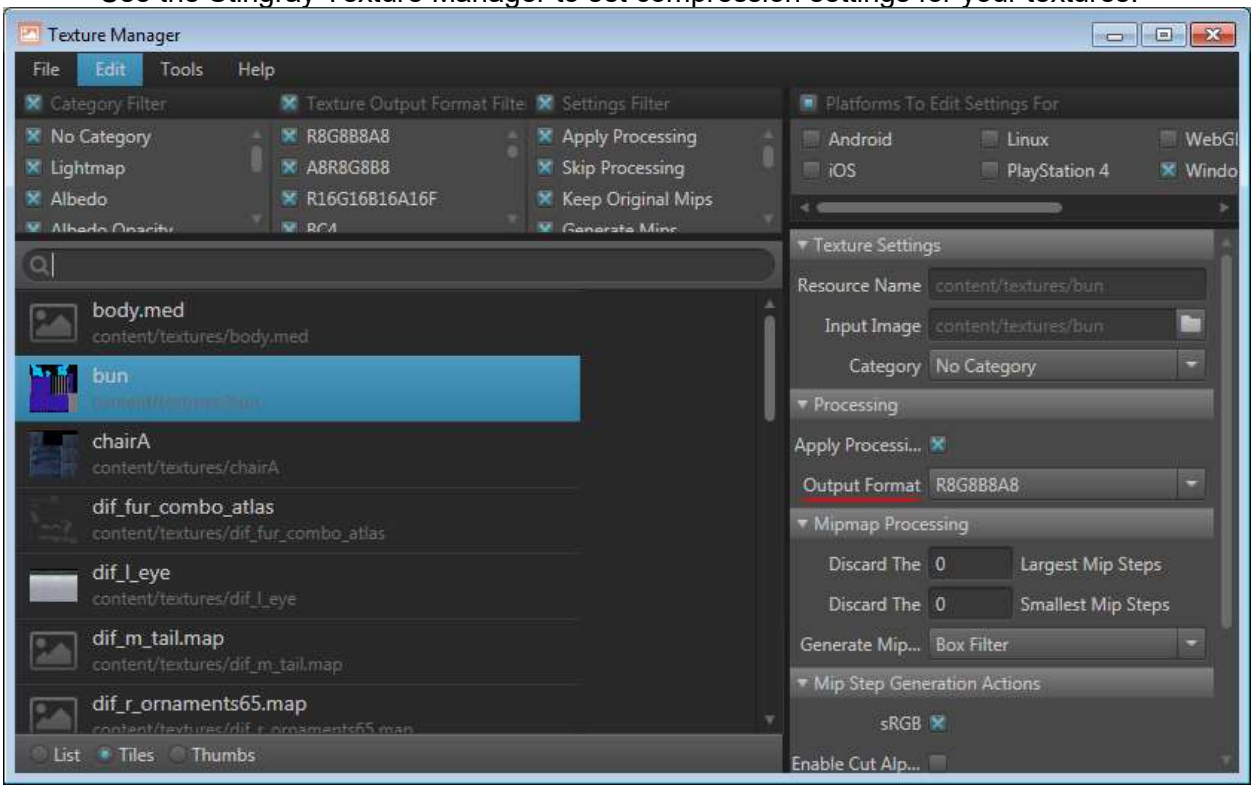
Now that all your textures are resized to reasonable dimensions, you'll want to compress your textures. Use the compression settings appropriate for the type of texture.

For textures in the standard import and RMA graphs use:

- Color (albedo, no alpha): DXT1
- Color with alpha: DXT5
- Normal: BC5
- Roughness, Metalness, AO (single-channel linear textures): BC4
- RMA: DXT1

Tool: Texture Manager

Use the Stingray Texture Manager to set compression settings for your textures.



TO OPEN THE TEXTURE MANAGER, DOUBLE CLICK A TEXTURE IN THE ASSET BROWSER, OR BROWSE TO IT THROUGH THE WINDOWS MENU.

Output Format
In the Stingray Texture Manager, look at Output Format in the Processing rollout. The default setting is R8G8B8A8 – which is uncompressed 8bits per channel including an alpha channel.

Mipmap Processing

Note: Textures can also be resized here by discarding the largest mip-map steps. This is useful because it is non-destructive to the original texture and can be specified by platform.

Shadow Casting

Shadow casting in real-time rendering is expensive. We'll cover some of the details of why they are expensive below, but to begin with you can make simple changes to reduce the amount of shadow casting that is happening per frame.

First, to determine if shadow casting is a problem use the Artist Performance HUD (perfud artist command).

Tool: Artist Performance HUD

ENABLE PERFHUD ARTIST TO EVALUATE RENDERING TIME SPENT CASTING SHADOWS.

The first bar in the performance HUD is for Shadow Casters. If it's red, then you have a problem with shadow casting!

Reduce Shadow Casting

Reduce shadow casting where possible. For example, you can disable mesh shadow casting in the following cases:

- Objects that are fully in shadow do not need to cast shadows
- Objects whose shadows are never going to be visible such as floors

Merge meshes:

- Meshes with the same material can be combined to reduce draw calls
- If materials are not the same, this will have little effect

Convert Point Lights to Spotlights

- Spot lights are 1/6th the cost of point lights. More on this later.

Create Shadow Proxies

High polygon assets do not necessarily need highly detailed shadows. For these assets you can create shadow proxies.

- Duplicate the mesh and reduce poly count, or create a simple low poly mesh cage
- Set proxy to invisible but enable shadow casting
- Disable shadow casting on the visible mesh

Bake lighting

If your scene is mostly static (objects do not move, lights do not move) then you can bake lighting for most lights.

You may need a sun light for casting shadows because real-time shadow maps will yield sharper shadows.

Batching

When the engine prepares to render a surface, it dispatches draw calls to the GPU in batches. These batches consist of data like vertex buffers, shader code, etc. With this data, the GPU driver then renders the surface to the screen.

There is some amount of overhead in the dispatching process. When we try to render too many batches that overhead will start to build up lowering your performance.

Reducing shadow casting is in effect a problem of reducing batch counts.

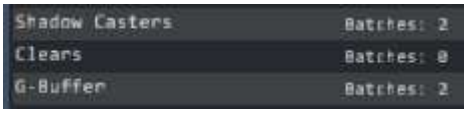
For any given frame, there will be one batch per material, per mesh, per shadow casting observer, per camera.

(b)BATCHES = (m)MAT * (g)MESH * (s)SHADOWCASTER * (c)CAMERA

b = m*g*s*c

For example, a scene with one spot light, a cube with one material, and a plane with one material will be:

2 material * 2 mesh * 1 spot light * 1 camera = 4 batches



But, we usually need sunlight in the scene, so let's switch to a directional light.

Switching to a directional light means we now have a sun light and we are using cascaded shadow maps.

The sun is used to light the whole world, so it renders higher resolution shadows closer to the camera, and lower resolution further out from the camera.

The sun has three cascade maps, and therefore has three shadow casting 'observers'. Therefore the sun light adds three batches whereas a spotlight only adds one.

You can imagine the sun light as a camera from which it observes shadows – each cascade is a single observer.

Now, let's move the cube away from the camera a little ways. At some point the batches for this box will increase by one:

The increase is because it is unfortunately between two cascades – and is visible in two cascade light observers.

If you have a very expensive asset, such as a distant city scape for example, adjusting the bias of the cascading shadow map to ensure that this asset lies in only one cascade is one way to increase performance.

Now, let's consider a point light:

You can imagine a point light as a cube, with a shadow casting observer pointing in each of the six directions of the cube. This is why a spot light is 1/6th the cost of an Omni-directional point light. The spot light only has one observer for the point light's six.

Reducing Batches

Reduce materials

Each material on a mesh increases its batch count by one. When possible, simplify the unit to a single material.

Shadow proxies

Again, another reason to use shadow proxies is because the shadow proxy of a unit with many materials can be reduced to a single material, reducing draw calls.

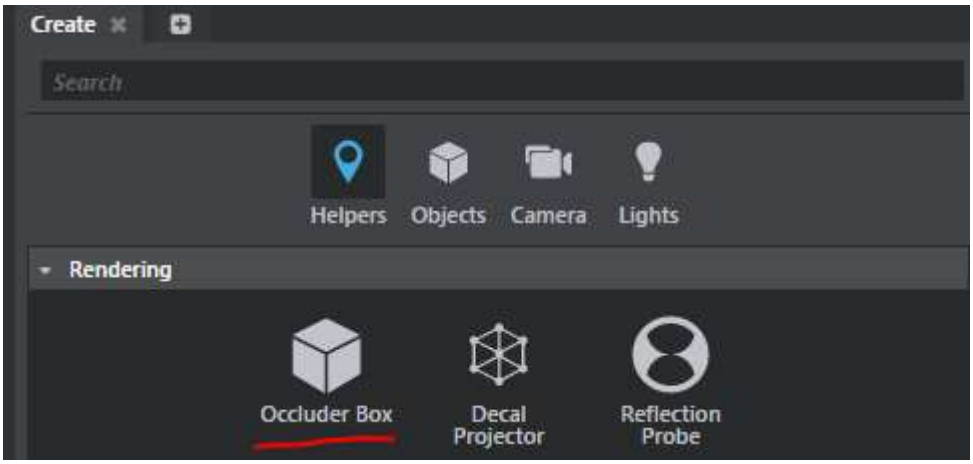
Occluders

Occluders can reduce the amount of primitives visible from any observer. It's important to realize that occluders occlude not just from the camera – but from ANY observer. This can have some unexpected results.

For example, if your occluder is larger than the visual mesh, you may get some unexpected shadows occluding from the sun light if you are not considering how the occluder affects the sun shadows.

Occluders come at a cost – therefore they are always cubes and any mesh set to be an occluder will use its bounding box to occlude.

While any unit can enable occlusion on its meshes, an Occluder Box can be created using the Create Tool.



Batch Merging

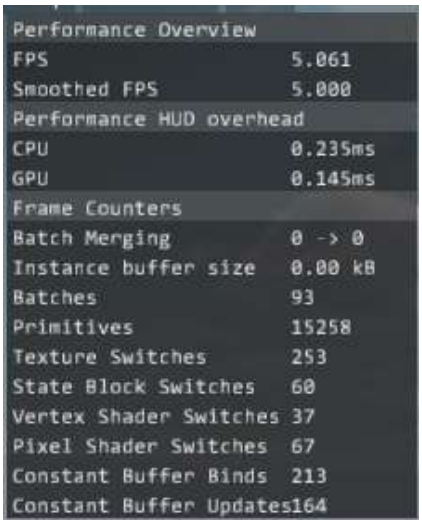
When the engine has multiple instances of the same geometry with the same material(s), and the material supports instancing, then these geometry instances can be merged into a single batch. This means that all the data for these objects get sent to the GPU in one batch.

Materials must support batching, and the standard import material does not.

Tool: Perfhud Artist

Perfhud Artist tells you what you need to know about the amount of batches that you are doing per frame.

The Frame Counters section of the HUD gives you information on batch merging, overall number of batches, and primitives.

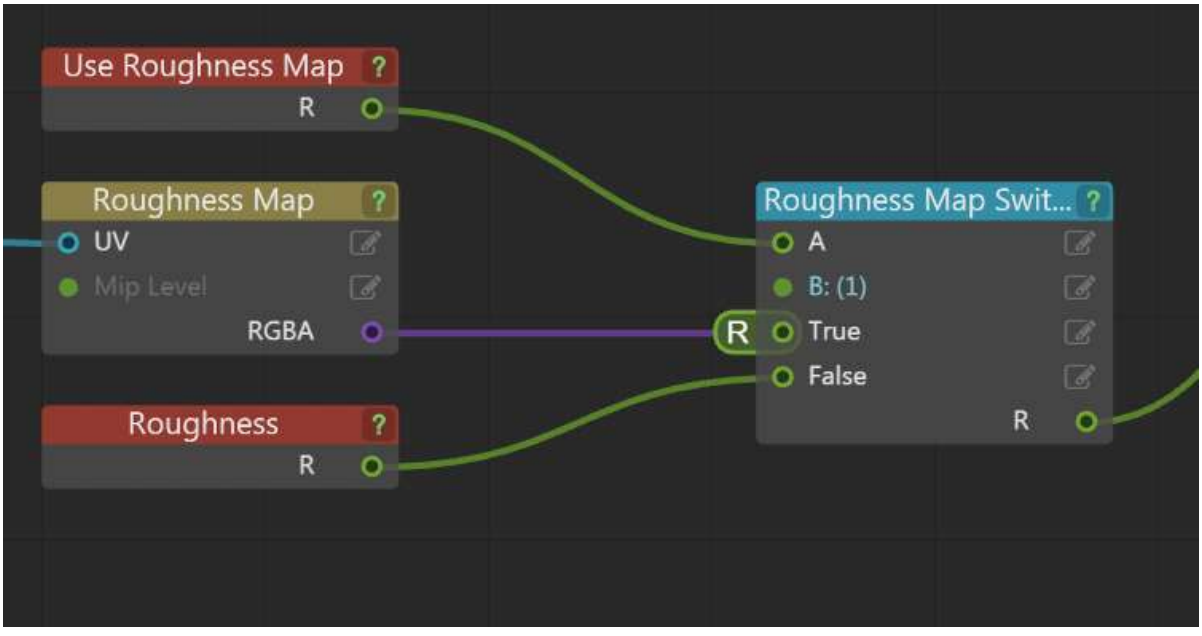


You can also see on the right the amount of batches/primitives per render pass.

Shadow Casters	Batches: 9	Primitives: 3452
Clears	Batches: 0	Primitives: 0
G-Buffer	Batches: 4	Primitives: 4532
Forward Opaque	Batches: 0	Primitives: 0
Opaque Lighting	Batches: 10	Primitives: 72
Reflection Probes	Batches: 1	Primitives: 12
Emissive	Batches: 0	Primitives: 0
Skydome	Batches: 1	Primitives: 4416
Transparency	Batches: 9	Primitives: 2538
Post Processing	Batches: 57	Primitives: 228
- SSR	Batches: 28	Primitives: 112
- SSAO	Batches: 6	Primitives: 24
- TAA	Batches: 1	Primitives: 4
Scaleform Studio	Batches: 0	Primitives: 0
Present	Batches: 0	Primitives: 0

Material Optimizations

The standard materials are optimized for compatibility – not performance.
Each sampler (texture), each switch, input, etc., comes at some cost. To get the most performance out of your scene you may also need to optimize your materials and replace the standard material with simpler materials.

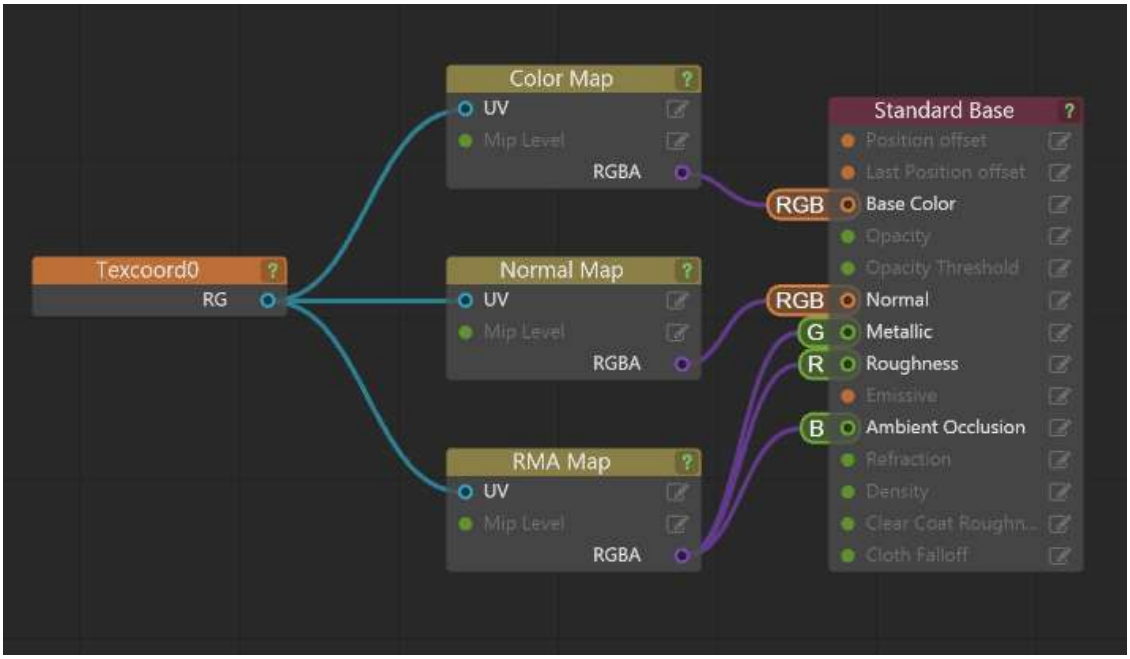


Standard_RMA

Stingray v1.6 comes with standard RMA materials. RMA stands for Roughness, Metalness, and Ambient Occlusion.

The standard RMA material has no switches or inputs other than a Color Map, Normal Map, and an RMA map. Because it has no inputs that need to change, you can enable instancing for this material.

The RMA material requires all these textures, and can take some effort to convert existing materials.



Construct Simple Parent Materials

Finally, if RMA materials are still too complicated for you (for example, you may not have any AO or Metalness maps) then you can easily construct your own and quickly convert existing standard materials to use your new material.

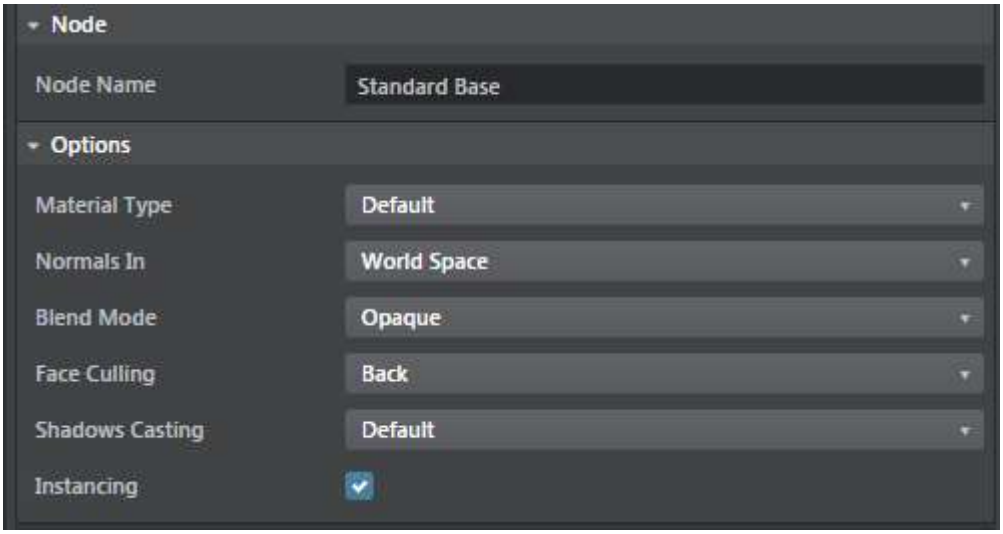
Start with a standard material and convert to a unique material by clicking ‘Make Unique’ in the Property Editor.

Remove any unused texture samples, and any unused switches.

Save your material – call it something like “simple_parent_mat”.

Now go to any existing material and change the parent material to your new material. If your material has matching texture sample inputs, then those texture input connections will be preserved.

If your simplified material has no inputs – or all the objects using the material have the same inputs – you can enable instancing for this material. Select the Standard Base node in the material graph and turn on the ‘Instancing’ property in the Property Editor.


Node
Node Name
Standard Base
Options
Material Type
Default
Normals In
World Space
Blend Mode
Opaque
Face Culling
Back
Shadows Casting
Default
Instancing