



FDC127489

## Using the Max Interactive Engine and Forge to create powerful AR/VR Applications

Benjamin Slapcoff – Autodesk, Inc.

Paul Kind – Autodesk, Inc.

Cyrille Fauvel – Autodesk, Inc.

### Learning Objectives

- Learn how to use Max Interactive to create useful, immersive AR applications
- Understand how modern real-time renderers work with AR and VR devices
- Understand more about Windows Mixed Reality and how to create intuitive workflows for working with your AR or MR devices
- Learn about how Forge can be used to power AR and VR experiences

### Description

This class will showcase the Max Interactive engine's augmented reality (AR) and virtual reality (VR) pipeline, and demonstrate how we can use Forge software to power collaborative AR/VR experiences on various platforms. We'll give an overview of how what the Interactive Engine is and how to implement efficient stereo-rendering pipelines in modern real-time renderers to use with today's AR and VR devices, as well as how to work with these devices in Max Interactive. Finally, we will showcase a demonstration using Forge to load models in an AR application on the fly.

### Your Forge DevCon Expert(s)

**Paul Kind** – Sr. Software Engineer

3D Artist and Game Developer who enjoys helping people come to grips with Stingray, Maya and Related Tools.

**Cyrille Fauvel** – Forge Developer Advocate

Cyrille Fauvel got his first computer when he was 12 years old, and as he had no money left to buy software, he started writing code in assembly code. A few years later, he wrote code in Basic, Pascal, C, C++, and so on, and he's still doing that. He's been with Autodesk, Inc., since 1993, joining the company to work on AutoCAD software originally. He's passionate about technology and computers. At Autodesk he's worked in various roles, from the design side to manufacturing and finally to games and films. He is now an evangelist for the Forge API (application programming interface) and web services, and he has a desire to deliver the most effective creative solutions to partners using these APIs and web services.

**Benjamin Slapcoff** – Software Developer

Benjamin Slapcoff joined the Autodesk Stingray/Max Interactive Rendering team full time after completing his B. Comp. Sc. degree at Concordia University in 2016, but he's been working on the team as an intern since 2014 on many different aspects of AR and VR in the Stingray engine. Recently, he has worked on adding support for the Microsoft Holographic devices and working on the integration with Forge. He has prior experience in the game industry before Autodesk, working at middleware, start-up, and more established game companies since 2009.

## High-level Overview of 3ds Max Interactive

### A Brief History

Regarding Game Engine technology at Autodesk you may have heard the terms Bitsquid, Stingray or Max Interactive. To avoid confusion, it is important to understand that they are essentially the same thing. Bitsquid was a Game Engine acquired by Autodesk in 2014 and rebranded as Autodesk Stingray. While the core engine technology was built upon and improved, the tools were almost exclusively rewritten. Stingray has since been integrated into 3ds Max and in this incarnation, it is known as 3ds Max Interactive. In this presentation we will be referring exclusively to Max Interactive.

### So what is a Game Engine?

Back in the era of the PlayStation 1 when people talked about a Game Engine what they typically meant was simply a renderer and the infrastructure to load the various meshes, textures and other resources to render them in real-time. However, things have evolved quite a bit since then and modern game engines include many components, for example:

- Rendering
- Audio
- Animation
- UI
- Physics
- Artificial Intelligence
- Inverse Kinematics
- Scripting (with the caveat that it needs to be fast, have a low memory footprint and run on any platform)
- Networking (LAN & WAN)

This is by no means an exhaustive list, so you can see that Modern Game Engines are quite complicated beasts. Moreover, a modern engine also includes all the various **tools** necessary to assemble the increasingly complex 2D or 3D worlds. Indeed, as content-creation team sizes continue to increase tools have become at least as important as the runtime they drive. It is also worth noting that while DCCs like Max & Maya are necessary to author much of the content, Game Engine tools are not equivalent to DCCs (at the most basic level, think of them more as a means to composite scenes from the assets authored in DCCs). Tools are thus an integral part of any modern game engine.



Figure 1 Some of the games made with 3ds Max Interactive

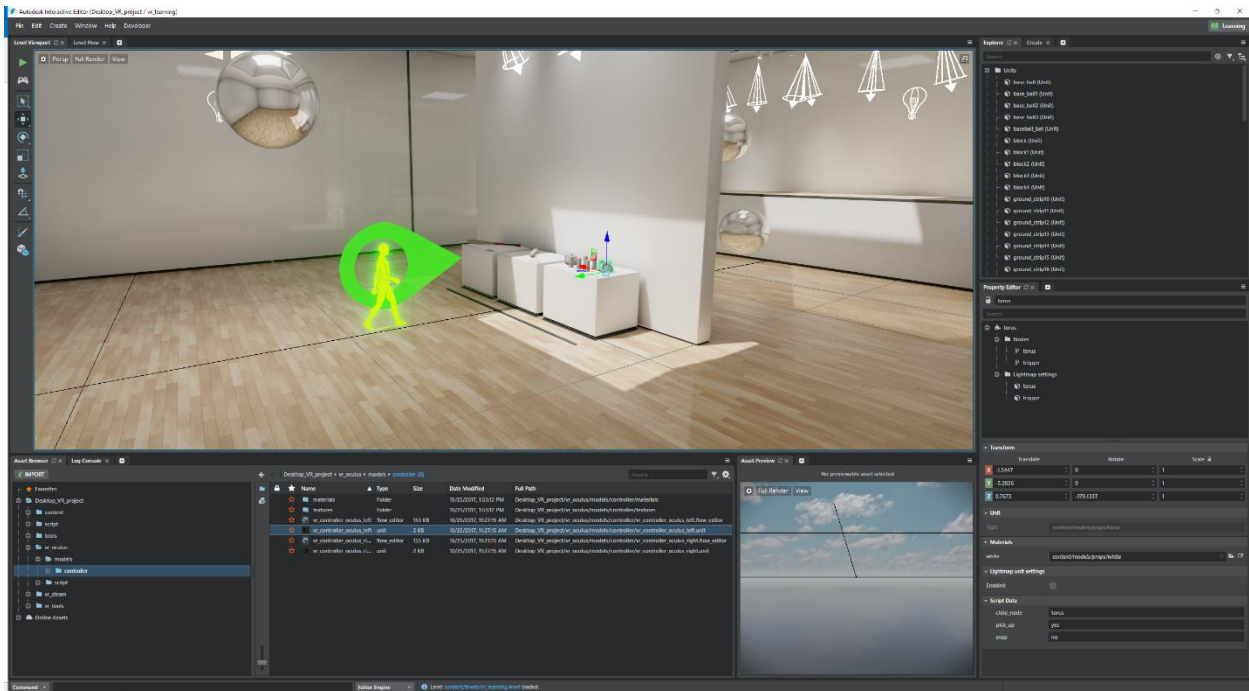


Figure 2 The 3ds Max Interactive toolset (known as the editor)

The various use-cases for Game Engines have also evolved significantly. Whereas previously game tech was only used to make Video Games, there are now many more uses with applications in:

- Realtime Visualization
- Cinema
- Animation
- Serious Gaming
- Simulation
- Training

Again, this list is not exhaustive. The advent of AR and VR has only served to increase the interest in using game engines in these domains to access the potential of AR/VR.



*Figure 3 Archviz in 3ds Max Interactive*

### **Key Design Goals of the engine**

The engine was designed with several key concepts in mind:

- Data Driven



# FORGE DevCon 2017

- The behavior of the Engine should not be hard-coded. Instead it should be controlled by data files that can be changed to produce a different behavior.
- Lightweight
  - Entire engine is around 400k lines of code
  - Makes maintenance easier and chances of understanding the code in general more likely
  - Having lots of code is not an advantage – it is a liability
- Quick Iteration Times
  - When you change something, you should be able to see the effects of that change immediately, on the actual hardware where your final game will run.
- High Performance
  - The engine must do a lot per frame (animation, physics etc.) in a very small amount of time. Framerate is especially important for AR/VR.
  - We are also running on a lot of different platforms with very different characteristics. Despite what smartphone manufacturers might want you to believe, no, your smartphone cannot compete with a desktop GPU...

## Supported Platforms

Modern Game Engines are expected to run on a multitude of platforms and 3ds Max Interactive is no exception it already runs on a wide range of platforms, including:

- Windows
- iOS
- android
- Oculus
- HTC Vive
- GearVR
- Daydream
- Cardboard
- Hololens
- Windows Mixed Reality
- ARKit
- Web
- Linux (experimental)
- macOS (experimental)
- more to come!

As you can see we have focused heavily on support for AR & VR Platforms and we intend to keep adding more.

## Features

As mentioned previously, a wide range of features are expected by users. 3ds Max Interactive includes:

### Data-Driven Renderer

The data-driven nature of the renderer means you can produce drastically different results with the same executable:





These are clearly radically different rendering styles, yet they are both produced with the same engine and no code modifications.

## Scripting

Max Interactive uses Lua to provide scripting support, primarily because it is lightweight, easy to learn and will run on any platform since it is written in C. LuaJit is used to further improve performance. In terms of writing scripts and debugging them we have a plugin for VS Code.

```
-- Returns the viewport position from the Editor, if available for example when
-- run as an editor Test Level.
function Appkit.get_editor_view_position()
+   if Appkit.boxed_editor_view_position then return Appkit.boxed_editor_view_position:unbox() end
+   return nil
end
```

## C-API

Having an interface in C is powerful because C can be wrapped by pretty much anything. If for example you do not want to use Lua as your scripting environment you can use the C-API to provide support for another language (indeed this is exactly what our Lua API does).

```
struct UnitCapi
{
+   ConstVector3Ptr → → (*local_position) (UnitRef, unsigned index);
+   CApiQuaternion (*local_rotation) (UnitRef, unsigned index);
+   ConstVector3Ptr → → (*local_scale) (UnitRef, unsigned index);
+   ConstLocalTransformPtr (*local_pose) (UnitRef, unsigned index);
}
```

## Plugin System

It is possible to extend both the runtime and the editor with plugins. On the runtime side of things we use the C-API to allow plugins to extend the engine.

Here are some examples of plugins:

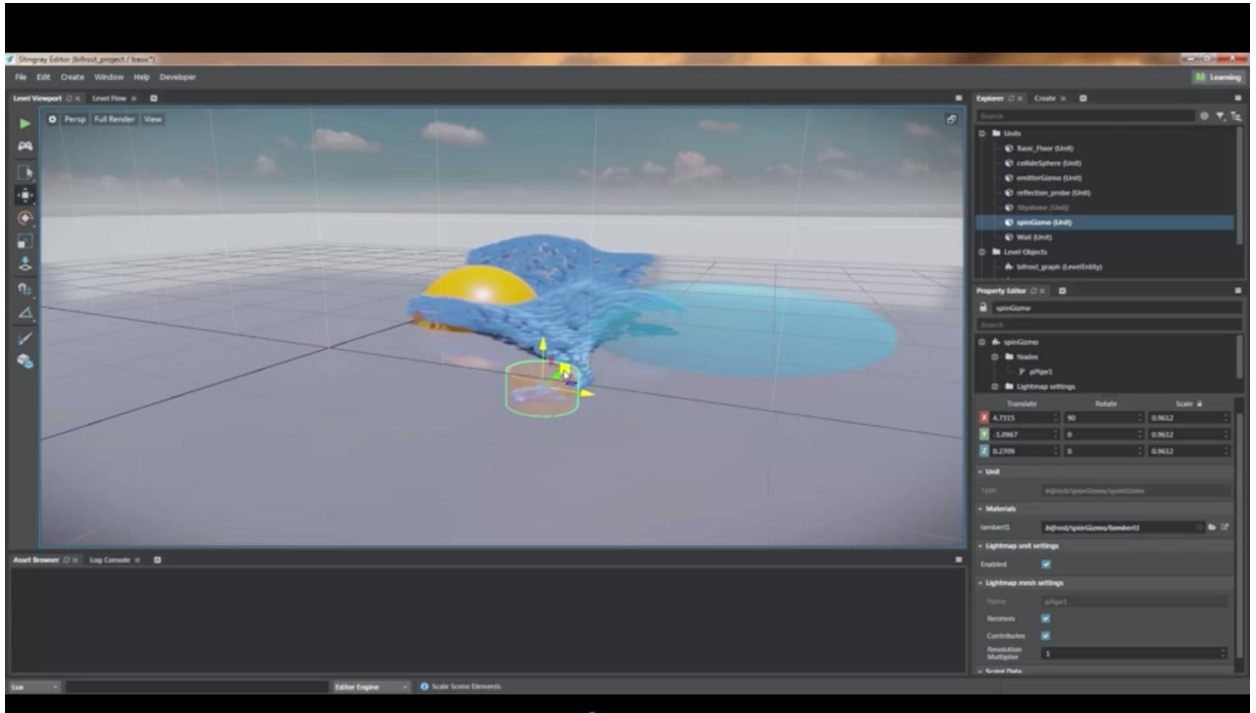


Figure 4 Volumetric Clouds Plugin



Figure 5 Occlusion Culling Plugin





0:56 / 1:59 1x

Figure 6 Bifrost Plugin

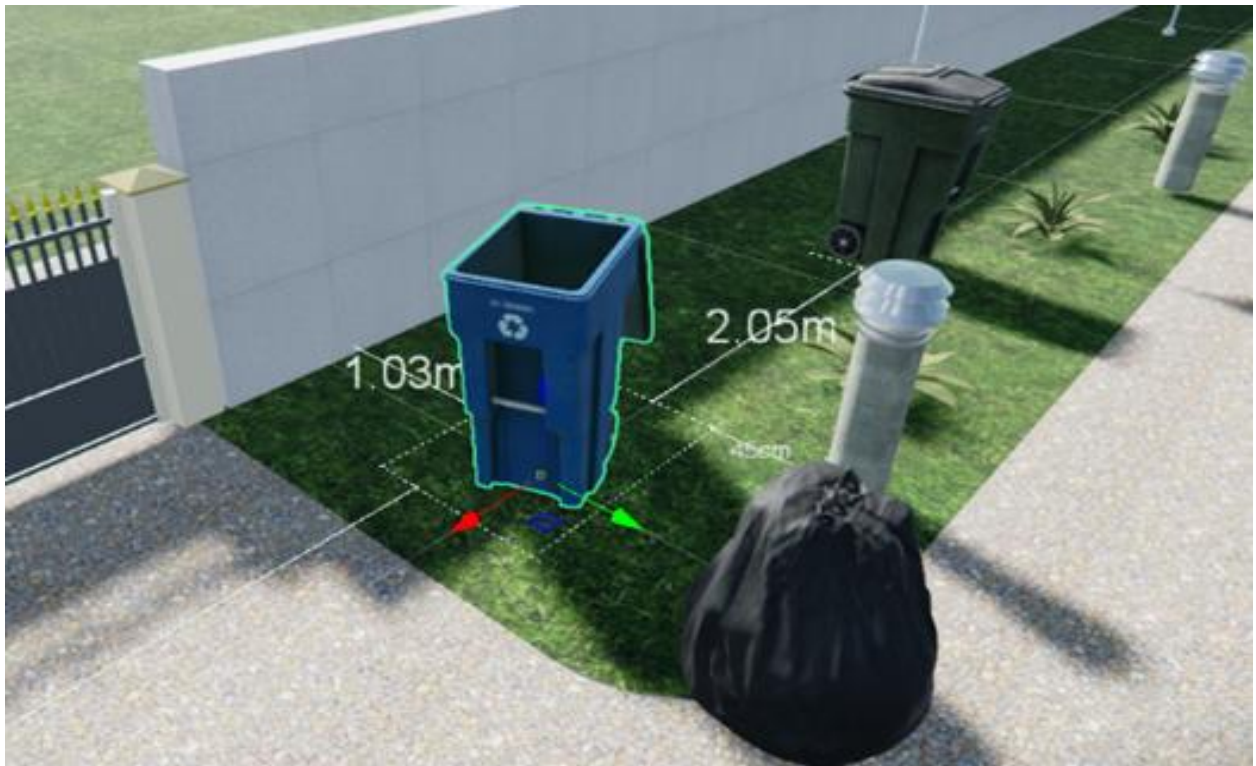


Figure 7 Smart-Placement Plugin

The Editor included a plugin-manager that allows the user to download existing plugins and have them automatically available for use.

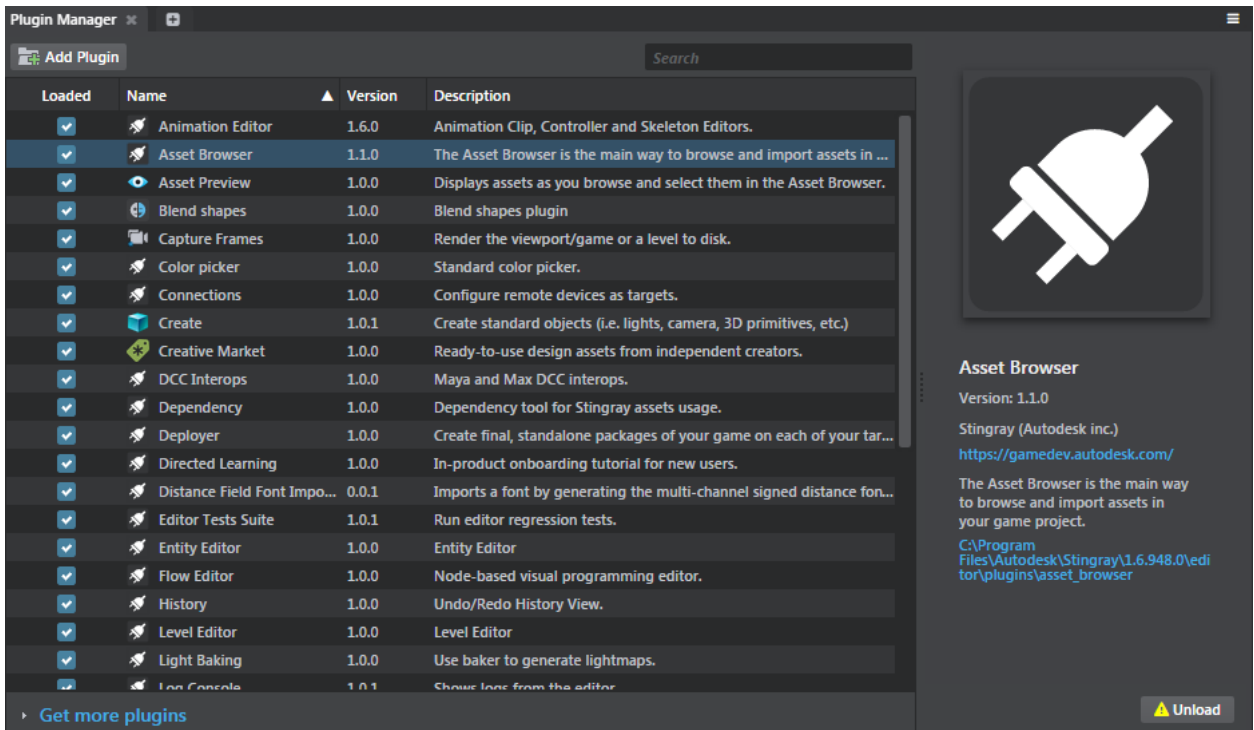
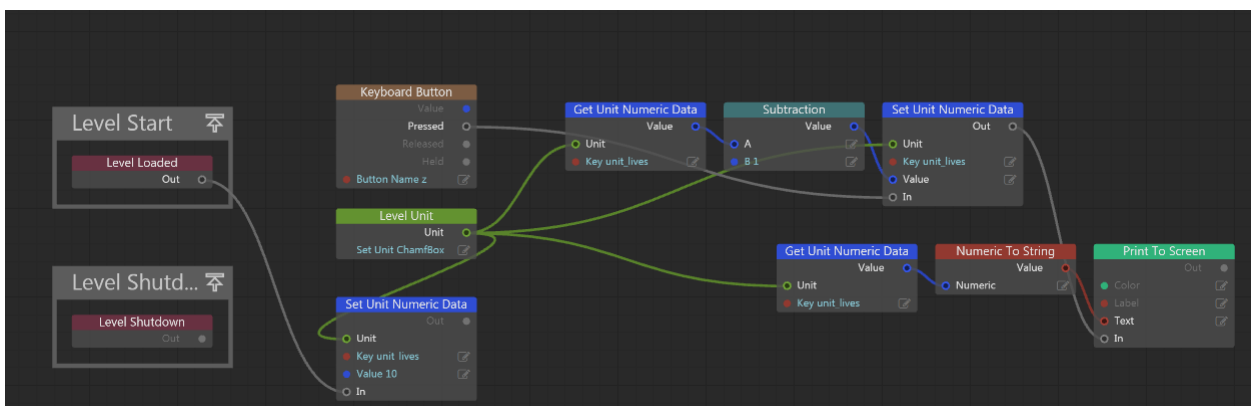


Figure 8 The Plugin Manager

The SDK necessary to write a plugin for the engine is available on github publicly and can be found here:

<https://github.com/AutodeskGames/stingray-plugin>

## Flow





# FORGE DevCon 2017

Flow is the visual programming system built in to Max Interactive. Flow allows you to author simulation logic without writing any code. Using Flow, you can set up scenarios, trigger effects, or define how units react to events in their surroundings. Flow is excellent for rapid prototyping or to allow the engine to be used by users who are not comfortable writing code.

## Physics

We use NVidia's PhysX SDK in order to bring our simulations to life physically. Features include discrete and continuous collision detection, raycasting and shape sweeps, solvers for rigid body dynamics, fluids, and particles, as well as vehicle and character controllers.

## AI

The engine includes a full integration of Autodesk's Navigation AI SDK. Navigation allows AI controlled characters to find and follow paths from place to place in the game world, avoiding collisions with each other and with other dynamic obstacles; and it provides spatial analysis tools for getting information about the virtual world.

- Automatic generation of Navmeshes
- Runtime PathFinding and Spatial Reasoning

## Audio

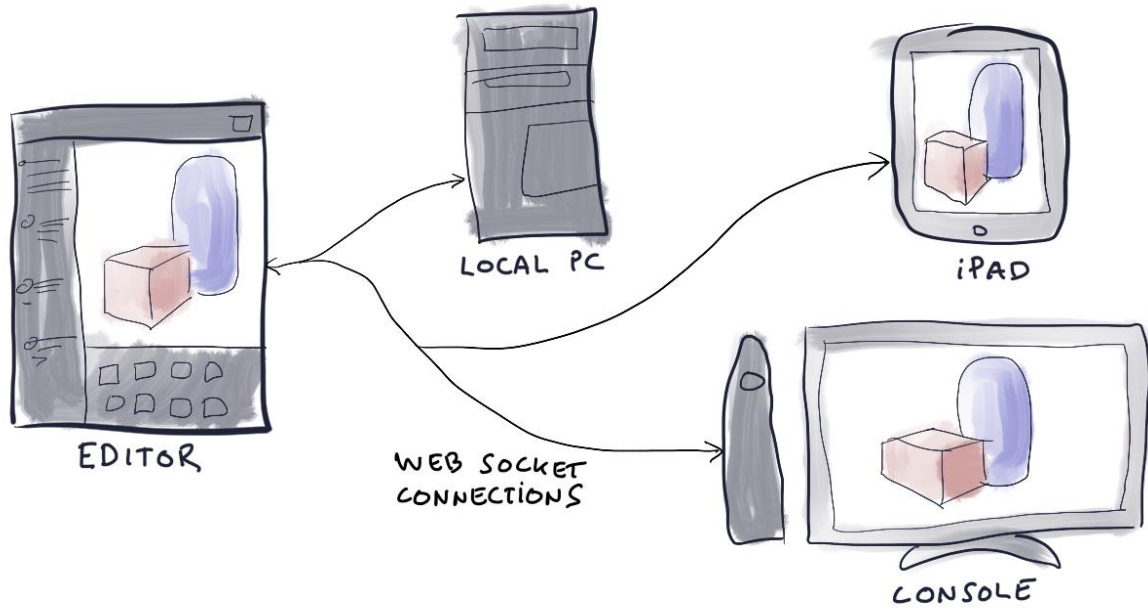
Audiokinetic WWise is an industry-leading sound engine for 3D engines, with a complete pipeline that includes editing tools for setting up content, monitoring and debugging playback, and live modification of sound properties.

## Animation & IK

The engine has its own animation system and also includes an integration of Autodesk's very own HumanIK. HumanIK is a full-body inverse kinematics (IK) solver and re-targeter that lets you create believable character animations, with effects such as foot-planting. look-at amongst others.

## Live-link and hot reload

Since rapid iteration times were identified as a key design goal for the engine we wanted content creators to be able to change things and instantly see the result on the target device. E.g. if they are developing for iOS rather than their workstation we want to see the result on the iOS device. This is achieved via a network connection to the target device and data-driven nature of the engine – changed assets can be hot reloaded without having to recompile the engine.





## Max Interactive's Rendering Pipeline

As illustrated by the few examples mentioned in the last section, Max interactive'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/AR/MR, 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 Max Interactive's 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. Max Interactive 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 if needed.

```
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

Max Interactive 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 Max Interactive’s default renderer is found in the install path under /core/stingray\_renderer/renderer.render\_config and is completely customizable without the need to recompile the engine code.

## Stereo Rendering Support in Max Interactive

Having a flexible renderer was key to implementing stereo support for AR/VR/MR devices. It’s important not to underestimate the many challenges in terms of rendering that stereo setups 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:

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

For common resolution types running at 60Hz we observe that stereo rendering is more demanding than running a 4K application at native resolution.

Resolution	Refresh Rate (Hz)	MPixels per second
720p	60	55
1080p	60	124
2160p	60	498
<b>Stereo (3240x1800)</b>	90	525

## Built-In Stereo Rendering Optimizations

To improve general stereo rendering performance, three major enhancements built in to Max Interactive have been put in place.

### Compound Frustum

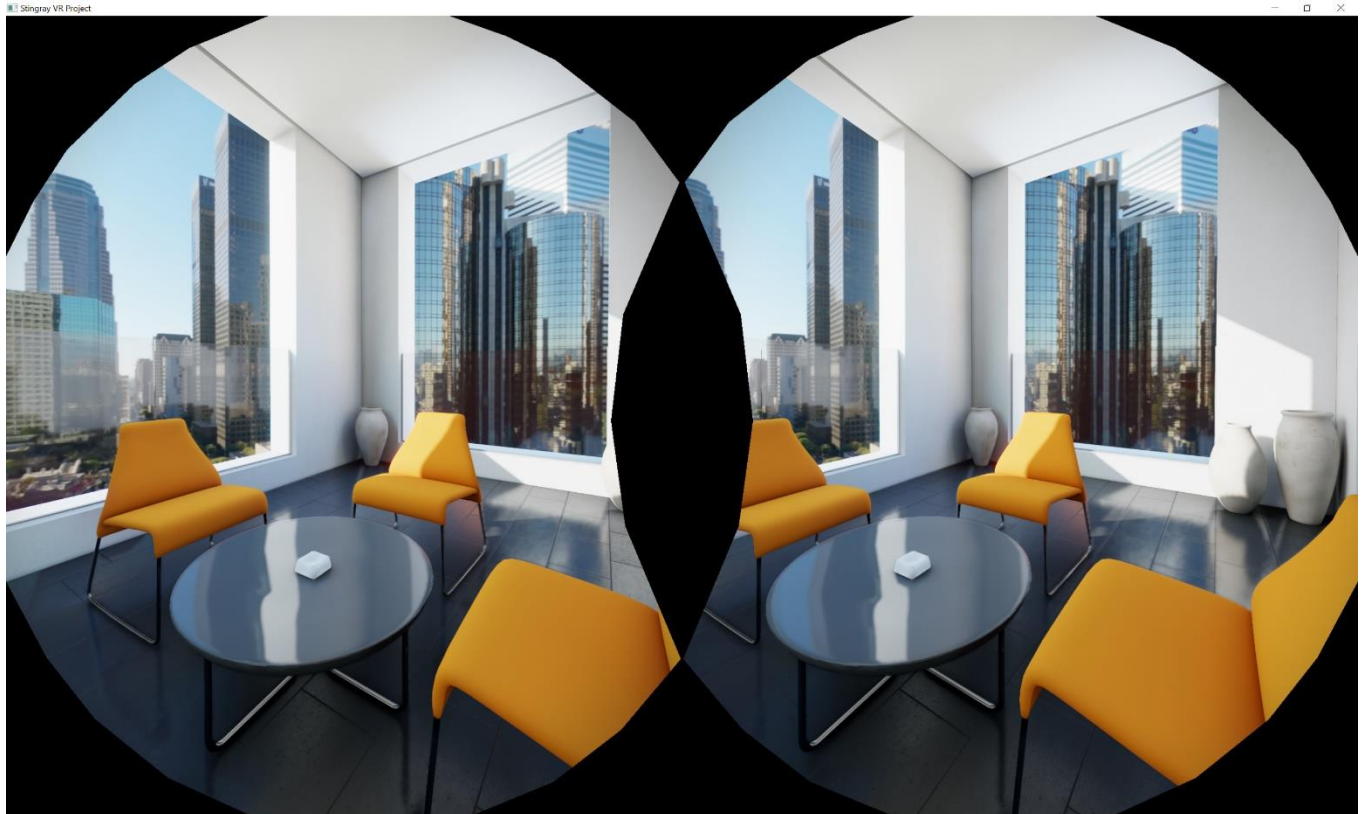
The camera frustum is used during different stages of our pipeline. First, we use it to optimize our rendering by performing frustum culling. This step gathers and prepares all



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 stereo 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.

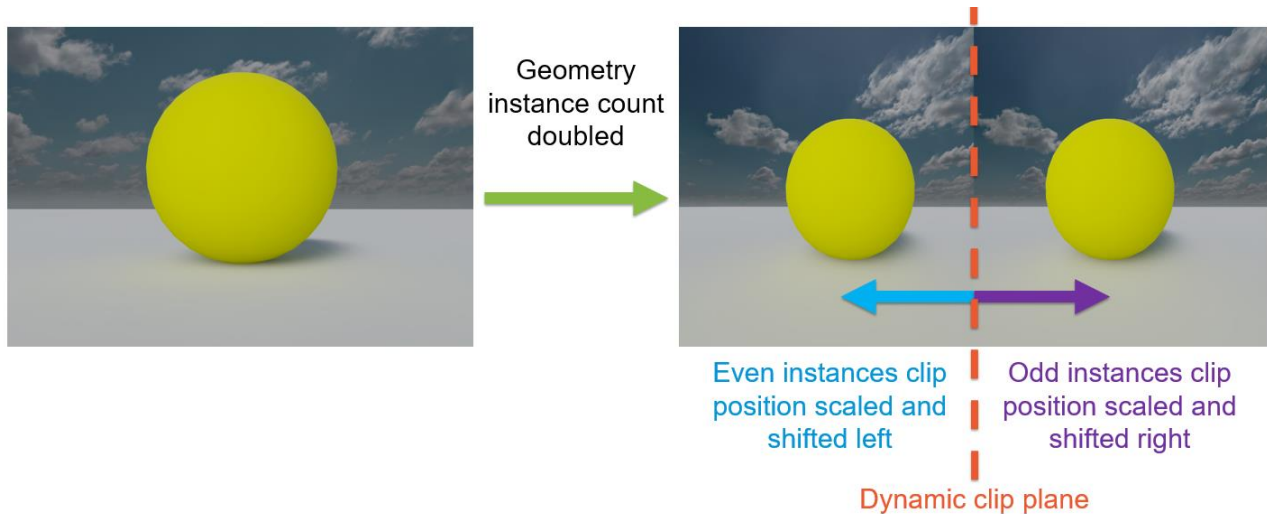


BLACK PIXELS REPRESENT THE HIDDEN AREAS ON THE HMD



## Instanced Stereo Rendering

Instanced Stereo Rendering is an optimization that makes it more efficient for Max Interactive to render stereoscopic images for AR/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.



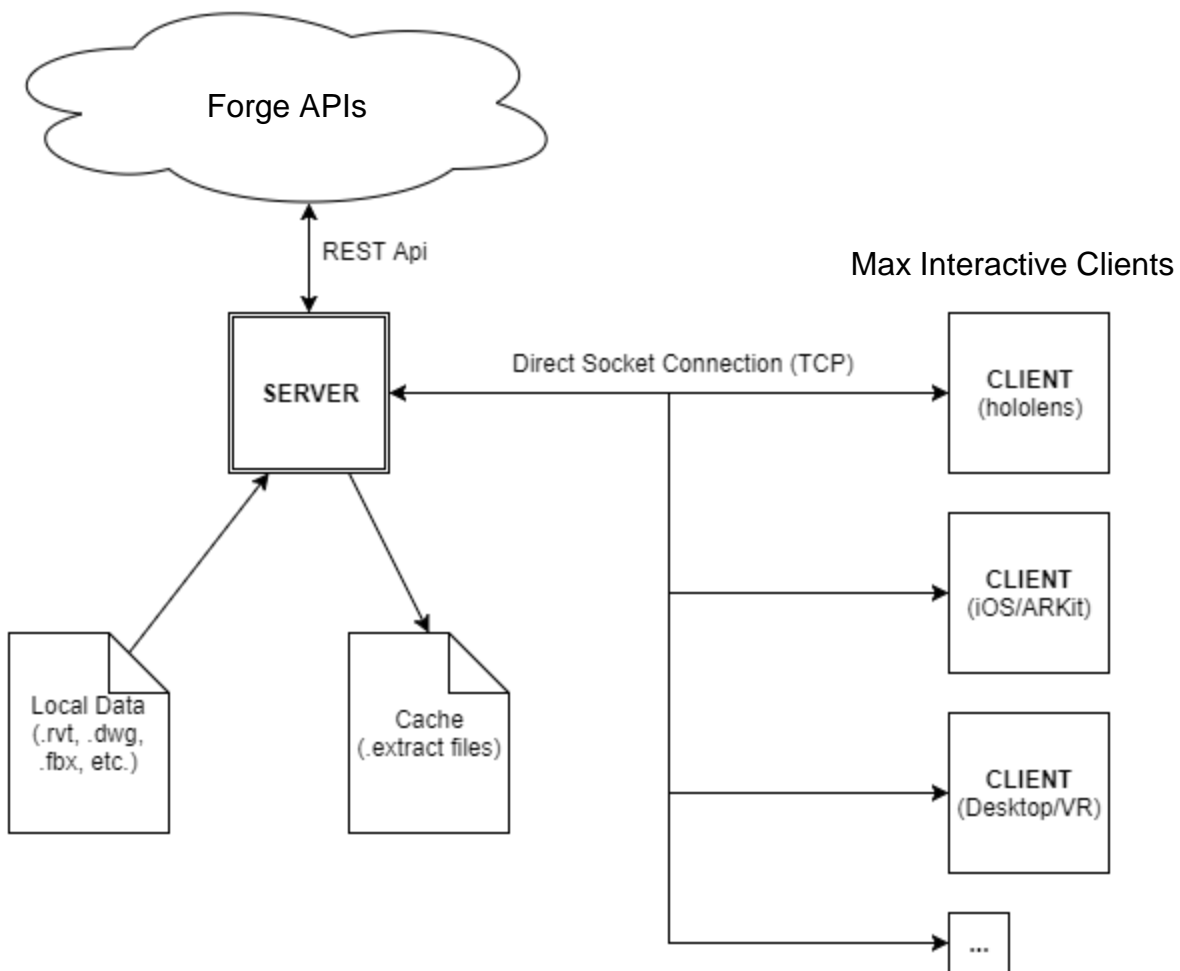
*INSTANCED STEREO RENDERING PIPELINE*

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 built in to the default renderer provided with Max Interactive. All material and post effect shaders have been stereo enabled to implement this optimization.

## Forge AR/VR Toolkit with Max Interactive

Now that we've gone over what the 3ds Max Interactive Engine is and how our rendering pipeline does its job, it's time to get into how we actually use Forge. For the first pass at our Forge plugin, we've adopted a Client/Server architecture. This enables us to have a server running as a desktop application that interfaces with the Forge services in order to prepare our data for any incoming client. Clients can be any platform the Interactive Engine supports, which includes AR (HoloLens and ARKit in our case), VR (GearVR, Google Daydream, Oculus, Vive), as well as basic desktop, Android and iOS platforms. Future iterations of our Forge plugin might alleviate the need for this architecture setup, but for now it serves its purpose.

This diagram shows the Forge plugin setup:



*CLIENT/SERVER ARCHITECTURE DIAGRAM SHOWING HOW THE INTERACTIVE ENGINE INTERFACES WITH FORGE*

## Forge ARVR Toolkit: Server

The first part we'll dive into is the Server application. This is the standalone application that interfaces with Forge and sends data to incoming clients. As the server application boots up, it reads in a config file for some initial setup called *server\_config.json*. Its contents look something like this:

```
{  
  "version":1,  
  "client_id":<client-id>,  
  "secret":<client-secret>,  
  "data_folder":"C:/TEMP/data/",  
  "cache_folder":"C:/TEMP/cache/",  
  "temp_folder":"C:/TEMP/temp/"  
}
```

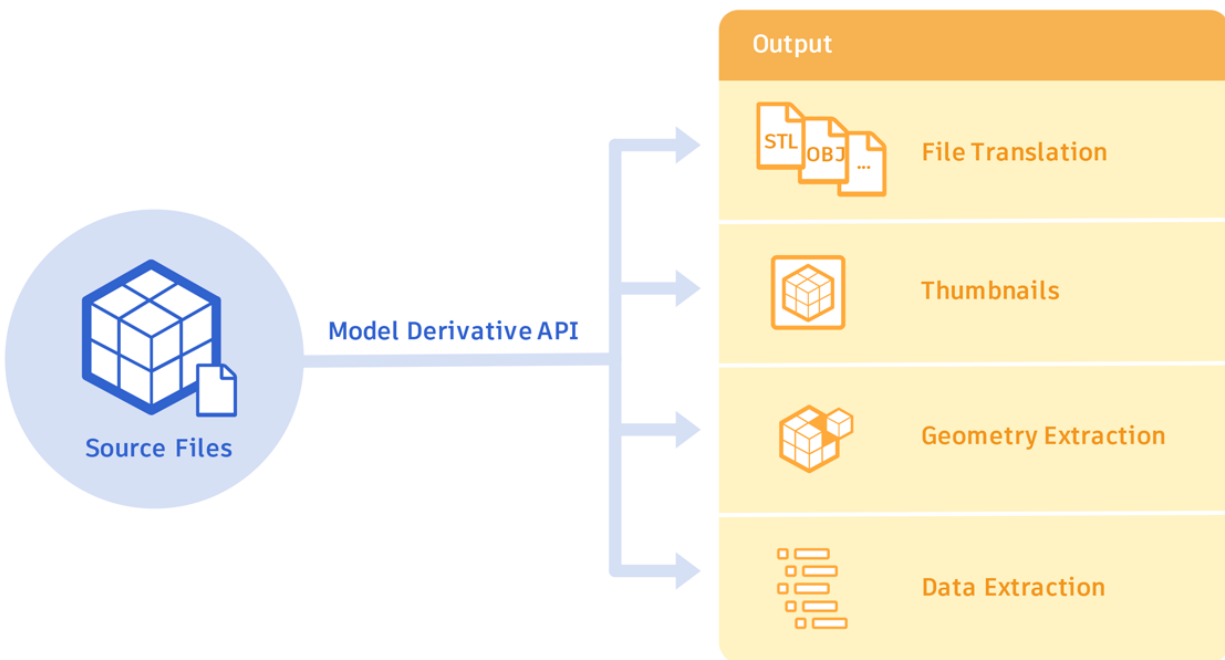
The client ID and secret are the IDs acquired when registering an app on Forge development portal (*developer.autodesk.com*), and the data, cache and temp folder paths are all used in different parts of the process, as will be explained.

Once the config has been read and parsed, we first go through every item in the **data** folder and register it as a resource we want to be available on the server. These are local files that will be uploaded to the Forge Model Derivate service, and can be of any file type that is supported by this service. There are over 60 supported file types, including file types such as .3ds, .dae, .dwg, .rvt, .obj, .fbx, etc. The full list of supported file types can be found here:

<https://developer.autodesk.com/en/docs/model-derivative/v2/overview/supported-translations/>

Next, we need to obtain an access token from the Forge OAuth API that will be used to make calls to other API endpoints. Using the client ID and secret defined in the server config, we perform a two-legged authentication request and store the access token for future use. The first request we need the token for is creating buckets in the Data Management API for each of our registered resources. These buckets will be used to store the objects we upload in the Forge Object Storage Service (OSS). Each bucket is created with a unique identifier for each resource, and has a "transient" retention policy, which specifies that the object should only persist for 24 hours. We then perform another request to actually upload our resources to these buckets, and store the object ID retrieved from the response as a Base64-encoded URN.

At this point we are finally ready to start interfacing with the Model Derivative service (the Forge Model Derivative API). The Model Derivative API enables users to represent and share their designs in different formats, as well as to extract valuable metadata. This API can quickly translate any resource into different formats, such as STL or OBJ, but in our case we want to translate the resource into the SVF format. The SVF format is normally used for extracting data and for rendering files in the Forge Viewer, but we'll extract the information necessary from these SVF files and then prepare it in a format ready for the Interactive Engine.



*THE FORGE MODEL DERIVATIVE API*

Using the URNs received from the OSS, we post a translation job to the Model Derivative service to convert each file we've uploaded to buckets into SVF format. Since these jobs are asynchronous and run in the background rather than halting the execution of the app, we can convert all of our resources at the same time. However, this means that once all our jobs are posted, we must poll for the completion of each job. Luckily the Derivative Model API exposes a method to get the current progress status of each job, so that is not a problem.

The Model Derivative jobs are done, but we are not ready to download the results just yet. First, we use the Extract API to extract the SVF into a "viewable" format, known as bubbles. These are the files we will download and transform into our own binary format ready to be consumed by Max Interactive. So we go through a similar process of posting a job to the Extract API and waiting for the jobs completion. Once all jobs are done, we download the results in the **temp** folder (specified in the server config).

The SVF files we now have on disk are zip files containing all of the relevant data for the Forge Viewer, including manifest data, fragment lists, light lists, camera lists, information about the hierarchy and instance tree, protein materials, geometry data, texture images, etc. We need to read in the relevant information and convert it into a format that will be easily consumed by the Max Interactive clients. We use two libraries to get the required data from the SVF files: the LMVTK (Large Model Viewer Toolkit) and the PropertyDB reader library. We unzip the zip files, find the SVF path and look for any “PropertyDB” file (these DB files contain a lot of metadata and information about the object). Then we start the conversion process.

The goal is to convert the SVF model into our own “Scene Model” representation, which is a model that is shared between the client and the server. The Scene Model is basically just a collection of scene elements, which each map to a node in the SVF file we are parsing’s instance tree. We start off by creating an LMVTK package using the path to the SVF file. We can fetch the metadata from this package which contains things like the unit measurement type of the object we’re reading in (useful for determining what scale factor to use, as it is meters in Max Interactive), as well information about the objects coordinate system (so we can map it’s up/forward vector to the Interactive engine’s).

This package also gives us an assortment of different readers (FragmentReader, InstanceTreeNodeReader, GeometryReader, MaterialReader, etc.)

- We use the FragmentReader to store all of information about each geometry node, such as its transform, geometry reference (used later with the GeometryReader stream), material entry, etc.
- The InstanceTreeNodeReader is useful for building up the scene hierarchy with our “scene element” representation. We traverse the instance tree by reading in one node at a time; if it is an inner node we build a “pure transform” scene element (i.e. no geometry at this node), otherwise if it is a geometry node we store the fragment information of the fragments associated with that node. In any case we also keep track of the nodes’ IDs and parent IDs to be able to correctly represent the hierarchy.
- For each fragment node we build we also want to fetch some data from the property database. To do this we need to open the “objects” table from the property DB file which was in the SVF zip, using the PropertyDB reader library. From this we can store any metadata associated with that fragment (name, attributes, etc.) so that it can be queried at runtime on the client side.
- We use the MaterialReaderStream to build each of the material representations that are going to be used to build the actual Max Interactive materials on the client side. From the material reader we call *getSingleSimpleMaterialJSON* to get a JSON representation of the material, from which we can parse different material parameters



(diffuse color, transparency, roughness, etc.) We keep track of these properties as well as the material's ID, since each fragment refers to a material ID.

- Finally we use the GeometryReader to get the actual vertex data for each fragment. When we read the fragments, we got a "geometry reference". We must use this geometry reference with the geometry reader in order to get all of the mesh data. We also make sure to convert the vertex position and normal data to the right space based on the metadata we read at the beginning.

Once all of this is done, we have the SVF file's complete scene representation in memory. Our scene model has the ability to be serialized and deserialized on demand, and that's where the **cache** folder comes to play. Each resource, after being uploaded to the model derivative service to be transformed to the viewable SVF format and then downloaded and converted to our own scene model, is serialized in a binary format to a ".extract" file in the cache folder.

At this point, all Forge jobs and data processing tasks are done, so we can finally setup the server code to start listening for incoming client connections. We open up a TCP connection at a specified port and set up loop for handling any incoming connections.

## Forge ARVR Toolkit: Client

The server has all of our Forge data prepared and is ready to serve it to any client that connects to it, so all we need to get Forge data into Max Interactive now is to make Max Interactive a client. To do this, we've created the "ExtractPlugin" engine plugin as the client side of our Forge integration. Engine plugins are C++ plugins that interface with the Interactive engine through our plugin API, and we can even implement Lua bindings to expose some of the Extract plugin's functionality at runtime.

The first step towards receiving data from the SVF server is actually connecting to the server. A Lua call was exposed in our plugin's "ExtractIO" Lua namespace. In a project, users can call

```
ExtractIO.server_connect(<ip>, <port>);
```

The IP and Port parameters are optional; by default, it uses your local IP and the port defined on the server side. When this is called, a TCP socket is created and a connection to the IP/port is requested. The server side receives this connection request and creates a thread to handle this client. It then responds with the number of resources in its cache (i.e. the processed SVF files) as well as their names. The client receives this information and creates a list of available server resources. A client can then use the following Lua call to view the available resources:

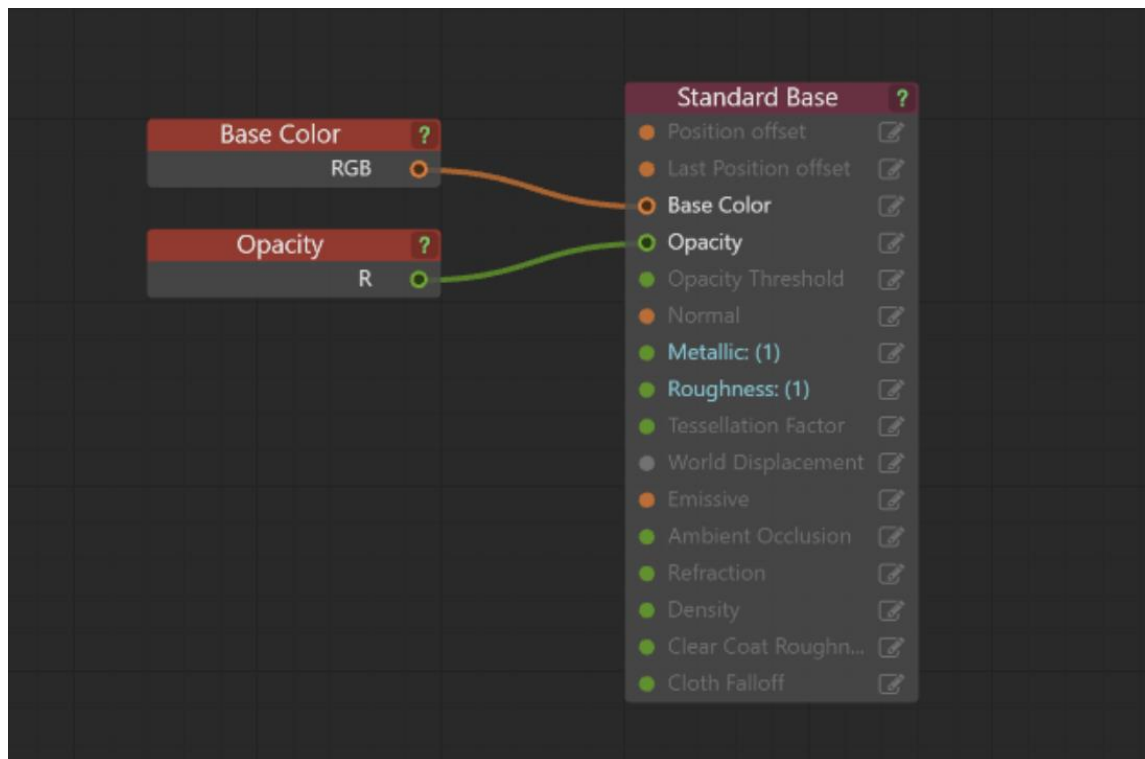
```
ExtractIO.server_resource_names();
```

This will return a string array with all available resources. To then request the actual data from the server, they can call:

```
ExtractIO.load_server_resource(<resource_id>);
```

Where *resource\_id* is the index of the resource in the resource list. This ID is sent to the server, and the server will respond with the size in bytes of the requested resource, followed by the resource binary. The client will receive this binary and convert it to the same Scene representation (i.e. collection of *Scene elements*) as was on the server side. From this scene representation we need to do another conversion into the actual data format that the Interactive engine can understand.

The *resource\_manager* class in the Forge plugin handles the calls to Max Interactive's plugin API to create the necessary engine resources to handle the Scene representation. Using the *SceneGraph* API, we create a scene graph node with the correct pose information for each Scene Element in the SVF representation. Some nodes which are "transform only" will only be a scene graph node, but scene elements that have geometry need to have their meshes created. Using the *Mesh* API, we can create a mesh handle for a mesh that will be handled by the engine and fill in its vertex buffer information using the *RenderBuffer* API. Then we call also use the material parsed from the simple material in the SVF file. In the project that will load the SVF file, we create basic material graphs with some parameters exposed, and then using the *Material* API, we can fill in those parameters with the values we have saved. Here's an example of a material graph with two parameters exposed for the base color and transparency:



MATERIAL GRAPH WITH TWO BASIC PARAMETERS EXPOSED

The last thing we create for each mesh is a physics representation using the *RuntimePhysicsCooking API*. This creates a PhysX mesh based on the mesh vertices so that we can do things like raycast against individual fragments.

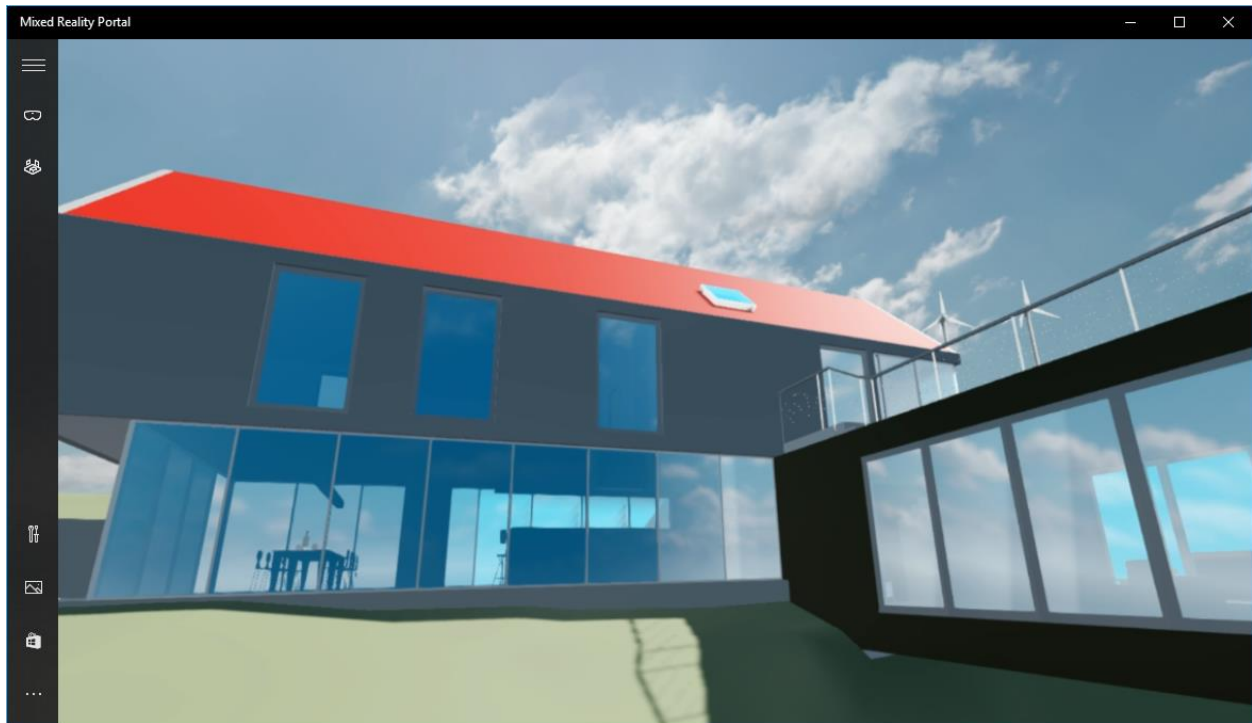
For each fragment, we also store any metadata that is present to be able to fetch at runtime and display if needed. With all of this, we have a complete SVF scene representation ready to be rendered at runtime in the Interactive Engine.

Other Lua bindings are exposed for convenience sake as well. Functions such as getting a fragment's bounding box or associated metadata are useful for different user interactions. These functions take in a node index, since when you raycast against the fragments physics object it will return the node index at which it is in the scene graph. This makes it easy to click on a fragment to raycast against it, get its scene graph node index and then access any information you want through the exposed Lua API functions.

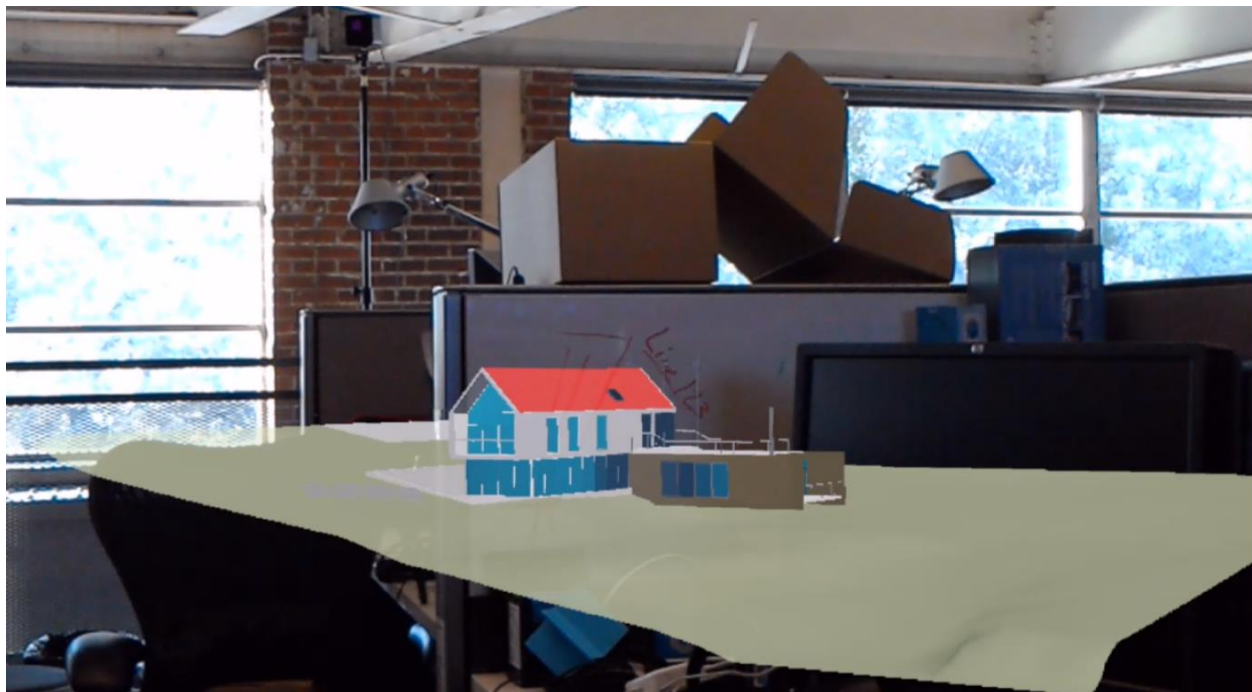


*FORGE AR|VR TOOLKIT RUNNING ON ARKIT*





*FORGE AR|VR TOOLKIT RUNNING ON MICROSOFT MIXED REALITY DEVICES*



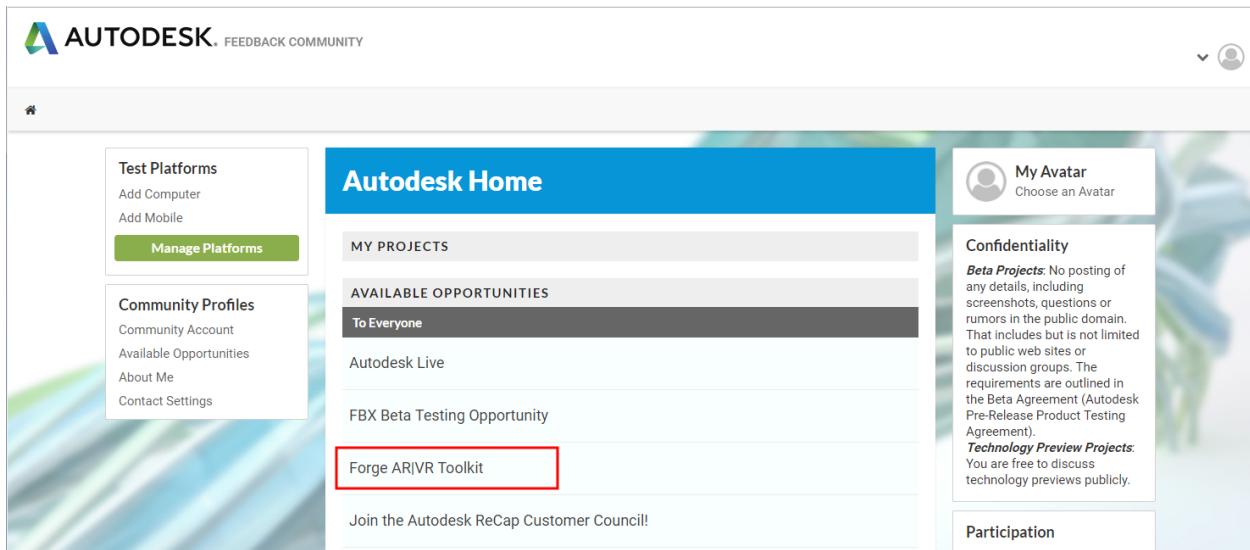
*FORGE AR|VR TOOLKIT RUNNING ON HOLOLENS*

## Get your hands on it

If you want to try out this workflow for yourselves, we're packaging up and releasing all the code for the SVF server and client projects.

Visit <http://beta.autodesk.com>, sign in with your Autodesk account, and look for an opportunity to join the **Forge AR|VR Toolkit** tech preview.

From there, you should have everything you need to get started. You'll also find forums to talk back to us about your experience and how you'd like to see us improve and extend the solution.

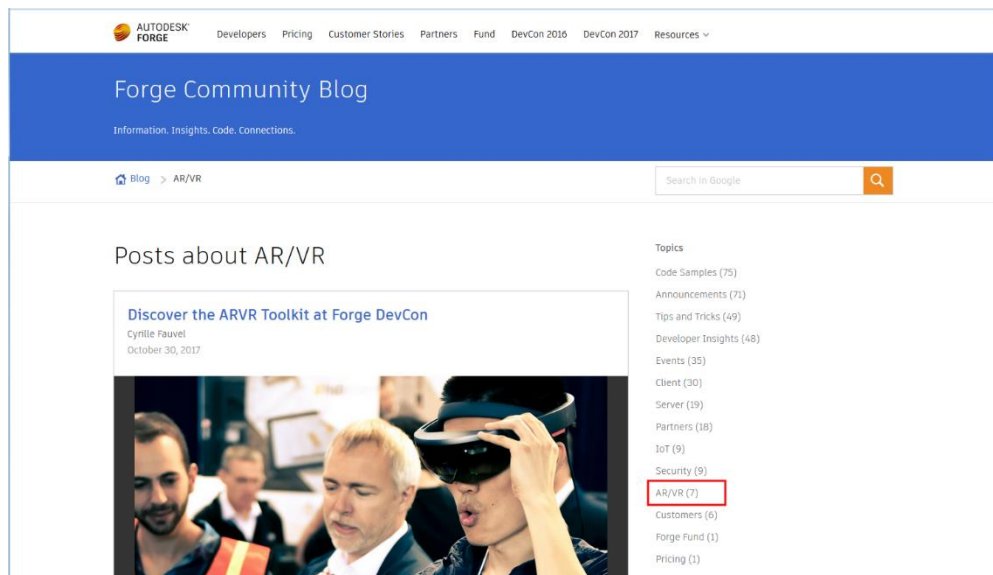


The screenshot shows the Autodesk Feedback Community homepage. The header includes the Autodesk logo and 'FEEDBACK COMMUNITY'. The main content area is titled 'Autodesk Home' and features several sections: 'MY PROJECTS' with a red box around 'Forge AR|VR Toolkit', 'AVAILABLE OPPORTUNITIES' with a sub-section 'To Everyone', and 'Autodesk Live'. On the right, there is a 'My Avatar' section and a 'Confidentiality' section with text about beta projects and technology preview projects. A 'Participation' section is also visible at the bottom right.

## Stay tuned for more

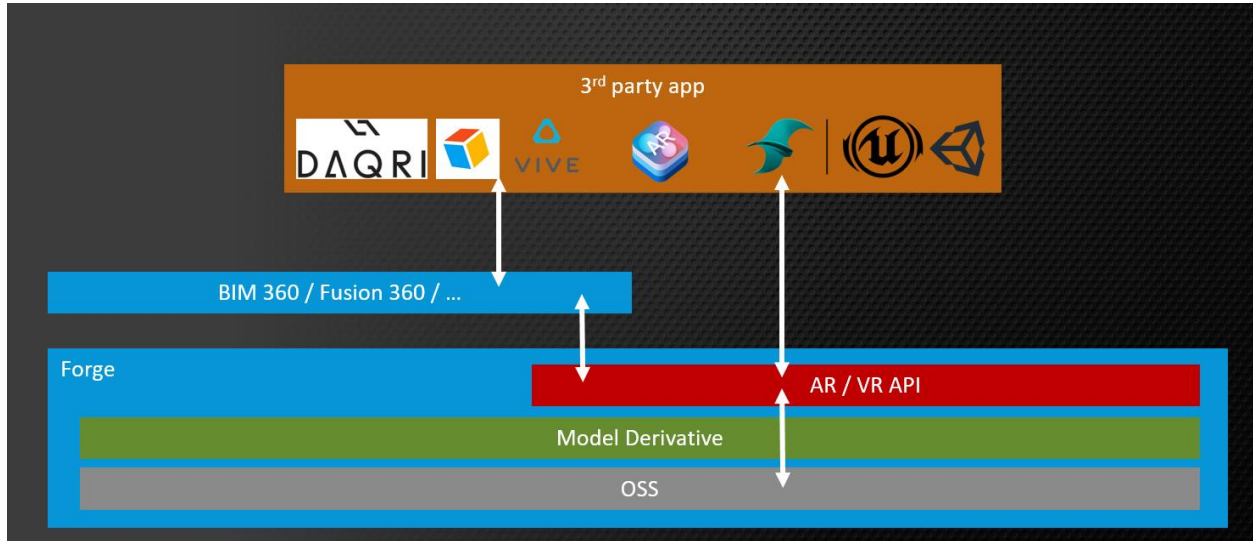
Also, keep your eyes on <https://forge.autodesk.com/blog> -- as we work on the toolkit we'll be posting all our news there.

Use the **AR/VR** tag to zero in on just the posts about our toolkit.

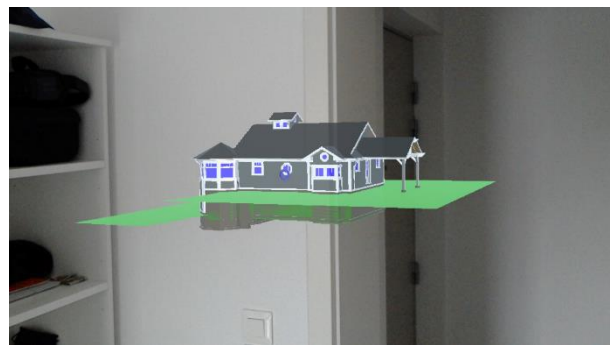


The screenshot shows the Forge Community Blog page. The header includes the Autodesk Forge logo and navigation links. The main content area is titled 'Forge Community Blog' and features a search bar and a list of posts about AR/VR. A red box highlights the 'AR/VR (7)' topic in the right-hand sidebar.

## Forge Demo Details



The public Beta of the new Forge AR/VR Toolkit will be demonstrated exclusively at the 'Forge Reality Playground' exhibit in the Forge 'Village' on Monday November 13th. Come and play with our live demos and talk to one of our Forge experts about how you can start using the Toolkit in your own workflows.



## **Acknowledgments**

**James Park** – Sr. SW Engineer, DCP-IXG

**Olivier Dionne** – Sr. SW Development Manager, DCP-IXG

**Raman Grewal** – Software Engineer, DCP-IXG

**Anis Benyoub** – (former) Software Engineer, DCP-IXG

**Robb Surridge** – Principal Learning Content Developer, DCP-IXG

**Matthew Carpenter** – QA Manager, DCP-IXG

**Daniel Cotnoir** – Director, Product Development, DCP-IXG