Virtual-Reality Procedural Experiences with 3ds Max’s Max Creation Graph and Stingray

Louis Marcoux B.Eng, BA
Autodesk Canada

Learning Objectives

- Create procedural content with 3ds Max's Max Creation Graph (MCG)
- Author real-time VR in Stingray
- Enhance VR experiences by triggering events for the viewer
- Setup fast updates and alternatives for rides review
- Decorate real-time environments using animated assets

Description

In this class, you will learn how to approach creation challenges in a procedural manner. We’ll use the subject of a roller coaster ride creation in order to see how fast it becomes to iterate when procedural tools can be built to help model complex systems such as roller coaster models. We’ll also see how to create VR experiences from the roller coasters that we’ll create. We’ll see how fast we can experience, change and experience again the rides, allowing for quick in context turnarounds at the design level. We'll use 3ds Max's creation graph to create procedural models and animations then we’ll use Stingray to build the VR experience. And since experience is key in VR, we'll study some tricks to enhance the environment with various types of animated assets.

Your AU Expert

Louis Marcoux is a senior solution specialist for 3D animation and visual effects at Autodesk Canada since 2003. He specializes in workflows for the games, films, and visual effects markets. He can also cover the various visualization industries when it comes to rendering, real-time interactive experiences and animation.
How did this idea begin?

Before I jump into the meet of the subject, I want to introduce the class by letting you know how it all started for me and how I got to writing this class.

In August 2015, one of my colleagues asked me to show a technology demo at the Games CTO Council at Siggraph. He wanted to show how procedural race tracks generation could help games companies to save time as they start to build a new race level.

He said to me, I want a system that:

- draw roads quickly from simple inputs
- populate surroundings of roads
- trees, blocks, lights
- iterate quickly
- flexible

You can watch the following tutorial in the “Tutorials” folder (see annex for download link) to see it in action and how to set it up:

01 Race Track Toolkit - Setup and Usage.mp4

This project was made of a lot of trials and errors to end up with a system that works. I was learning the tool and I was trying to fit my way of thinking to how the tool works. When project was over, I did a post mortem to better understand what I did right and wrong…
I ended up with big spaghetti of nodes in the graph and it was hard to read back and find out where things are. A project like this needs to be more elegant and better organized. It needs to be made of simpler components from which we can make sense.

![The High Level Graph for the Race Track Toolkit](image)

Most of all, I needed to start building reusable components and avoid copying and pasting nodes that would do the same thing. This kind of thinking would save a lot of time in future projects. Also, the nodes would need to be a bit more generic so that they can be more flexible and adaptable to other projects.

I could go in depth about functions and logic but I want this class to be made of easy recipes that you can put together to create bigger systems.
Procedural Content

Procedural content can be defined in many ways. In the context of this class, I'll define procedural content as a system of simple things that come together to create a meaningful result. Leaves and trunks can create a tree, many trees of multiple species can create a forest and many forests and land can create an island. Those are systems that we can make a sense of.

Trees, Plants, Forests can all be Procedural

In the context of 3D modeling, procedural content can also be an single object that you can describe with meaningful parameters. Parameters that a system can interpret to build the object so that you don't have to model it manually. A hand can have 5 fingers, an electric pole can have a transformer and wires, etc. So many things could be modeled by only entering a few parameters and let the system figure it out.

An Electric Pole is Made of Simple Things put Together in a System
When you build these objects and systems, you need to use various techniques to arrive at your final result. You can start with simple meshes and put them in relation to each other. Clone them in a meaningful way, transforms each occurrence in a matter that makes sense for the system.

You can define some creation logic with conditions. You can use functions and algorithms to shape a mesh and make use of your college mathematics to put it all together. The main goal of all this is to create a tool that will be so simple that it will make your life easier when you need to build something that could be repetitive. Also, it will allow you to iterate quickly by only changing a few parameters to get different results.

So, building a procedural content system also means to create a simple tool that will have an impact at a large scale. You can change the look of a forest, change the species, define height variations, age, color, etc. Instead of having to open multiple interfaces and edit meshes individually, you can adjust a few sliders to do so.

The Max Creation Graph (MCG)

The Max Creation graph is the tool I used to create procedural tools. It’s inside of 3ds Max and it complements very well the modeling toolkit to allow for more complex system creation. By connecting visual nodes together with a wide variety of function, you can create tools that will simplify bigger tasks. It’s not a wire parameter workflow with live updates in the viewport. With MCG, you compile tools for later use in 3ds Max.

You can also create packages that can be distributed and installed on other 3ds Max instances. In a graph, you build tools from inputs that can be nodes in a scene or parameters like any other parameter in the 3ds Max user interface.
The graph can output multiple things. Those outputs are available in the 3ds max User Interface once they are compiled. Like any other plugin. You can create primitives, modifiers, splines and animation controllers. Even more if you look at the entire technical list of outputs.

![Outputs from MCG](image)

The rest of the nodes (about 900) are there to offer a multitude of functions to build and modify the data for the final output. It’s extremely powerful. We won’t cover it all in the class but we’ll use it to build our tool. It’s easy to get lost when doing MCG in front of crowd so, during the class itself, I’ll insist on the concepts and how I build things. Then, from the video tutorials that I have recorded and this handout, you should be able to go back home and study all the graphs. My goal is to enable you to be able to understand the graphs that are provided in the class.

To get a quick introduction to how the Max Creation Graph works, please watch the following tutorial from the “Tutorials” folder:

02 MCG Basics.mp4

![Simple Box Clone Tool Built in MCG](image)
The Roller Coaster Project

Before I started the project, I went after some visual references. I browsed the internet until I found a typical representation of what a roller coaster is. That’s what I used as my starting point to define the problem at hand. My end goal is to build a roller coaster generator that can be driven by only a few parameters and be fully interactive.

Let’s look at the main components of the coaster as seen on the image I used for my study:

![Study of the Visual Components of a Roller Coaster](image)

For the system to be flexible, I ended up with the following conclusions:

- The tool needs 3 objects from the 3ds Max scene:
  - A spline that defines where the coaster goes (heart line)
  - A spline that defines where “up” is through the ride
  - A ground to stand on
- Then, I needed to be able to control the following
  - Rails with variable size and distance
  - Central beam with variable size and distance
  - Ribs of variable number and size
  - Variable amount of pillars
  - Pillars must not be in the way of rails
  - Pillars need sections, a stand and a ladder
  - Sections on the central beam with connections

Finally, when the tracks are done, we need to be able to animate a train and a camera on the tracks based on gravity and an initial traction.

Lots to do, let’s get to work!
Starting a Project: Best Practices

When any project that leads to complexity starts, it is important to start simple. Then construct more and more simple things and connect them together.

My main approach is to first identify the components to build. Make them as simple as I can and make them as flexible as generic as possible. I don’t want to have to rebuild things again and again so it is important to plan reusable components in the system.

In MCG, there is a tool that allows you to build your own nodes with your own inputs and outputs. They are called compounds. By building compounds, it’s easier to get organized and to build components that are meaningful. I started to use compounds ad nauseam and it became clear to me that they were extremely helpful in getting organized. And quickly, I realized that I was able to reuse many of them in other contexts.
The Max Creation Graph also allows you to create groups. Those groups have a title, can have comments and they can be colored. I used them to layout the graph so that I can easily find out what a group of node is for.

Groups are great tools to layout graphs in an orderly fashion.

So, those are my best practices advices before starting a project.

Start Small : Creating A Mesh

Before we start to create a mesh in MCG, let’s look at how a mesh is created in 3ds Max. We’ll take the case of a rectangle, made of 2 triangles. Everything in polygons is defined by triangles at the foundation.
A rectangle is made of 4 vertices and each triangle constitutes a face. So, in 3ds Max language, we have 2 faces. To describe this, we need to number our vertices. This will be important for later.

![Numbered Vertices Diagram]

Vertices are described by their position in world space. Let's say we want a 5 units rectangle that stands at the origin in the Z plane. We would define the vertices as this.

```
vertices
1: (0,5,0)
2: (5,5,0)
3: (0,0,0)
4: (5,0,0)
```

Then, we need to build faces from those vertices. The order in which we define the vertices for each face is important. It will define the direction of the normal of the face. So it will be visible in front of that direction. Counter clock wise makes the face points towards you. Or, if you prefer, the normal points at you. Clock wise makes the triangle face the other way, opposite to you.
We define faces with the vertices indices like this.

```
indices
1: (3,2,1)
2: (3,4,2)
```

But that’s not all! In polygon world, we also need to define the smoothing groups of each face, the material ID’s for each face. And, finally, for each vertex in the geometry, we need to define where they are in UV space so that we can get proper mapping coordinates. All this needs to be defined for all polygons.

In MCG, if we want to create a custom mesh, we need to take the same approach. But with the nodes at our disposal. You start with the basics, the create Mesh node. And you want to output a geometry since we want to create a custom primitive. The create mesh node needs the vertices definitions in the form of an array (we need 4 entries in our case) and the face indices in another array (2 entries for our rectangle). We can define those from functions, mathematical equations, parameters or just static values. This is where some of the fun is. But, for now, let’s just say we have static vector entries to define what we had in the previous slide.
From that mesh definition, we need to define the Smoothing Groups, again an array of 2 entries with static values input (not vectors this time, face indices as integer values). We also need to define the Material ID’s from the same fashion. And then, again, an array of 4 vertices positions in UV space to define the UV map channel. And this is the simplest case since we can go much deeper in UV mapping definition.

And this is for a simple rectangle. Are you tired yet? The simplest element in the roller coaster is the bolt primitive that I intend to build to connect rails and pillars sections. We would have to repeat this process for 36 vertices and 50 faces!
Start Small: Simplify Mesh Creation Tools

Looking at the challenge of building more sophisticated meshes than a rectangle, I concluded that I needed to find a better way to put a custom mesh in MCG. You can instance a node from the 3ds Max scene as an input to MCG but that requires the mesh to be in a 3ds Max scene. Therefore, the MCG tool loses its independence from the scene. I wanted to have all the custom meshes created by MCG to be stored as part of the MCG package. But, I needed to find a way to simplify this process.

I did a bit of research, and then, I found this node, that can take a string and transform it into an array based on a delimitation character.

I immediately thought that I could create a tool that lets me input vertices position all at once and convert them to an array of vertices. Then, I realized that I could do the same with Indices, Smoothing Groups, Mat' ID's, UV Map. So I built all of those components and packed them together in a single compound: the String to TriMesh node (LM_StringToTriMesh).

I could input all that data at once and then get the mesh out. It was a great idea but when you use the Constant node to type in that data, you realize that there is a lot to type and it’s prone to errors…
So I created a Max Script (macro script) that takes any mesh selection in 3ds Max, creates a string with all the information about that mesh in the format needed by the node and puts the string in the clipboard. That script can be added to a quad menu for easy access:

```
So I created a Max Script (macro script) that takes any mesh selection in 3ds Max, creates a string with all the information about that mesh in the format needed by the node and puts the string in the clipboard. That script can be added to a quad menu for easy access:
```

![Image of quad menu with Get Mesh Info highlighted]

“Get Mesh Info” Macro Script Puts a String Describing the Mesh in the Clipboard

The workflow to create a custom mesh in MCG representing a custom mesh goes like this:

1. Model the mesh and define UV’s, Mat ID’s and Smoothing Groups
2. Select the mesh in 3ds Max
3. Run the Macro Script
4. Paste in the Constant Node in MCG
5. Connect the Constant Node to the LM_StringToMesh

![Image of MCG node setup]

**Nodes Needed to Create the Bolt Primitive in MCG**

I have included the MCG node as part of the distribution files. You can see how to set it up and use it in the following video tutorial:

```
03 StringToTriMesh.mp4
```

![Image of MCG bolt primitive]

**The AU Bolt Primitive Made with MCG**
Systems of Components

Now that we know how to create a custom mesh easily with MCG, we can start building bigger systems by adding many simple custom meshes together in a single graph. Then, we can put some basic transformations between those meshes to create something meaningful. This is the base of how we can start making really cool things based on just a few parameters.

Let's take a look at the case of the roller coaster pillar.

The goal of how we can use the pillar later is to make it stand between the ground and the central beam of the rails. It's basically between 2 points: one on the rail and one on the ground, directly below the one on the rail. As a standalone primitive, we can setup the pillar to be created with only a few parameters:

To make this work, let's decompose the pillar into its components and define how these components should behave in relation to the basic parameters.
Base

The base object can be positioned at the base of the pillar and scaled so that it matches the height requested by the user.

Anchor

The anchor can be positioned on top of the base and scale to match the size of the base.

This is the magic of using procedural primitive creation: all this can be adjusted by only a few parameters and give very strong flexibility.

Bolt

The bolt can be copied as many times as needed. All copies can be scaled, positioned and rotated to their final location and size.
Pillar Pole

A simple opened cylinder can be used to create central pole for the pillar. It just needs to be positioned at the same place as the anchor and scaled to match the desired height of the pillar.

Section & Support Beam

Again, a simple cylinder (closed, this time) can be used to create the section joints and the top support beam. It just needs to be scaled, rotated and positionned.

Ladder

For the ladder, a repeatable pattern can be used. We can duplicate it to fill the space between the base and the top of the central pole. Again, scale and position each copy of the ladder pattern will give us a full ladder.
Extra Support

Below the main support, there is an extra support structure that can be scaled and positioned. It can also be rotated to have it on both sides of the main pole.

With all those components and by using mathematics to find the correct transformations for each simple object, we can reproduce the structure of the pillar in a single primitive. That is the true power of using MCG for parameterized custom primitives. But since it’s all based on transforming those primitive meshes, we need to take a closer look at how transformations work in MCG.

Transformations

Transformations are very important to understand in the context of MCG and they may not be exactly like you are used to in 3ds Max.

All the transformations happen in Object Space and they start where the object was created. So, (0,0,0) corresponds to where the mouse click happened for the creation of the custom primitive. From there, the MCG meshes are created at (0,0,0) in that object space.

The mesh is defined by vertices and faces like we was in the previous section. There are no notions of sub object mode like elements or polygons or anything like that. It’s a mesh as a whole. Always. Transformations that are applied to meshes in MCG have no concept of parenting as well. All transformations have their pivot point at (0,0,0) in object space. If you rotate the mesh, it’s around (0,0,0) even if the mesh is not centered at (0,0,0). If you scale, the pivot of the scale is also at (0,0,0).
If you combine 2 meshes or more, they become a single mesh and they are transformed based on the (0,0,0) of the object space. Meshes that are far from the origin with rotate around the origin. This causes a bit of headaches when you want to rotate something that is not centered at (0,0,0). Therefore, you can plan transformations to avoid this problem.

In order to avoid transformations issues, you need to plan all transformations very carefully. The order of transformation is key in the success of all transformations. So, in MCG, your need to follow this order:

- First, scale the mesh based on the proportions you are after.
- Second, rotate to get the final rotation you are after,
- Then, translate and put in the right spot.

If you scale first, you avoid scaling in wrong axis. It scales in pure XYZ for the created mesh. If you rotate before positioning, you rotate at the center of the mesh if it was created at (0,0,0). Once the mesh is scaled and rotated correctly, moving it to its final position will have no effect on the integrity of the mesh.

Useful Mathematics

You have no choice! You need to dig back into your college mathematics… When building rules for transforming objects, you will need to do some simple maths. Here are a few notions that you will need to review in order to move faster in building MCG systems. My goal is not to refresh them here but to give you a few cues on what you need to revisit…

The Rule of 3

This rule is quite handy when you want to find the right proportions in which to scale an object. You start from the original dimensions of the mesh and then you can define how it needs to be scaled to match the desired size.
Here is a link to review how the Rule of 3 works:


Vector Algebra

In many occasions, you'll need to figure out an angle between 2 vectors, the normal of a plane, the position of additive positions and more. For that, you will need to review all the notions you can get from vector algebra.

Here is a link to review some vector algebra formulas:

http://emweb.unl.edu/math/mathweb/vectors/vectors.html

Trigonometry

Again, to find a distance based on an angle or find an angle based on distance for rotations and translations, trigonometry is very useful.

Here is a link to review trigonometry rules:

https://www.math.hmc.edu/calculus/tutorials/reviewtriglogexp/

To understand how to create a system of meshes based on transforming multiple meshes in MCG, please watch the following video tutorial:

04 Building A System.mp4
Working With Splines

Why are splines important for this project? If you look at the roller coaster, pretty much everything corresponds to 2 concepts around the spline. The rails are shapes lofted along the spline and the ribs are cloned along the spline.

Our challenge in this is to take the data coming from the spline and convert it or translate it so that we can make sense of it in MCG. Once we know how to get positions along the spline and the final rotation at that position, we can start using those transformations with what we have learned so far.

From MCG, let’s look at the data that we can extract from the spline. We want splines to drive the whole system so we want to have them loaded in the graph as a node to work with. The coaster will be generated along the input splines.

So, from a spline, we can get the position at a specific percentage.

If we look forward a small percent, we can find the forward vector as well.
But, from a single spline, we don't have any information about where is “up”. If we assume that up is always Z up, then we won't be able to make twists and upside down sections on our coaster. This is why we can't think of a system with a single spline. We need two: one being a very close copy of the other so that if we look at the same percent, we can find the up direction.

And, using simple cross products, we can find the side (right) direction at that point. With all that, we have a full coordinate system from which we can extract position and rotation. Or, even more useful in the context of MCG: a transformation matrix.

This is what I used to find the final transformations at various points along the splines.

For that system to work, we'll need splines that match closely so I recommend using a reference copy of the first spline for the “up” spline. We can then add an edit spline modifier to define twisting and banking of the up direction.
To simplify the work in MCG, I created 2 nodes that will help with further tools development with splines. The first one (LM_TransformMatrixOnCurve) is getting a transformation matrix that defines the final rotation and position at a specific percent along a spline if the spline as an “up” spline to work with.

With that matrix and using the Map node, you can clone multiple times along the spline and all clones will be perfectly aligned on the spline and will respect the “up” direction. For commodity, I added a local offset position that will allow to move in all directions from the final transformation. This means that we can clone left, right, up, down, forward from the local coordinate system we have from the spline percent. Very useful to put objects along the track on the side.

Finally, I also created a Loft node that allows to take an input mesh, standing on the XY plane, and extrude it along the splines. This will be used to create the rails and central beams.

The MCG nodes are provided with the distribution files and you can learn how to use them by watching the following tutorial:

05 Working With Spline.mp4
Finishing the Tool

Now that we have everything in place, we can put all this together in a single MCG graph. We have the tracks, we have the pillars and from these concepts, we can start adding all sorts of maths and transformations to give more flexibility to the user of the tool.

![Final Components of the Roller Coaster](image1)

With Max Script, you can rearrange the UI, group parameters that were created in the graph. It helps making a user interface that looks good and is clean.

![The Roller Coaster UI Edited with Max Script](image2)

With the same ideas and concepts, we can create an MCG animation controller that transforms objects along the 2 splines like the clone. It’s simply assigning the transform matrix to the output matrix of the controller.

![Transformation Controller From Splines](image3)
This controller will be used for have train animated on track and cameras if you want to ride the roller coaster.

The final MCG tool is called RCTK (Roller Coaster Toolkit) and is provided in the distribution files. To learn how to use it, please watch the following tutorial:

06 Roller Coaster ToolKit.mp4
Real Time VR

Before we move into the technical aspect of building the VR experience, let’s have a word about the #1 rule of VR. Everybody that talks about VR will tell you that, in VR, you should never move the point of view. That it will get you sick. Well, if doesn’t start well for our roller coaster ride since it’s all about moving along the ride.

But, if you think about it, if we considered “being sick” a reason not to build roller coasters in the first place, we would have never got anywhere with it… So, that brings me to question the #1 rule of VR... At the origins of cinema, in the early days, the #rule about the camera was… Don’t move the camera. It will make people sick and make them uncomfortable. Those were the same people who walked out of a theatre because a filmed train was moving towards them. Well, someone named D.W.Griffith decided in 1904 to say “screw it” and he moved the camera for the first time in a film.

So we’ll say “screw it” too and we’ll make the point of view move. It’s all about making this exciting. But I have learned something as I was building this project. If you ride the roller coaster in VR without the train and the wagon, it doesn’t feel right. But, when you add that, it becomes much more comforting and enjoyable. So, my golden rule for VR goes more like this:

If you move the point of view, put the viewer on a platform and make the platform move.

In our case, the wagon will be our platform.

Stingray (V1.5) Templates

To create a real-time experience (not a 360 video) with the ability to move around (head and body) we need a game engine. For this class, I will use Stingray but Unity and Unreal will follow the same principles if that is what you are using.

In Stingray, your starting point is in Templates. To start a new project, you can use one of the 6 templates. For VR experiences, we have 2 templates, depending on your targeted hardware. In the future, I am pretty sure hardware manufacturers are going to have consolidated workflows for VR but now, we are back in the VHS/Betamax or HDDVD/Blu-ray years where all technologies need their own approach.

So use the Oculus template for Oculus Rift or, like in my case, Oculus DK2. And, for the Vive, use the Steam template. They have great examples of how to setup the headset and also how to take advantage of the controllers. I won’t focus on controllers today but we’ll still use them to trigger an animation later.
As of now, and this is changing extremely rapidly, the Oculus is “In place”. Either you stand or you sit. But you can't walk around a room and see things from different angles. This is true for all setups using the In Place mode. In these situations, we use the camera as the key point of where you will be in the VR World.

**The Oculus Rift Will Soon Have Room Scale – The DK2 Was In Plane Only**

With the Vive, you can walk around in Room Scale. For that case, we care about the base of the virtual space and this is this space (and where it stands in the VR world) that will define where you'll be in VR. We need to be a bit more careful when building experiences for Room Scale because we don't know exactly where the person will be in that space.

**The HTC Vive Offers Room Scale VR**

When I originally started to work on this class, I wanted to talk about performance in VR, the rules of VR and the use of teleporting to move around in VR. But, those subjects are becoming extremely wide and I think it’s better to let other classes focus on those. The AU line up offers quite a few examples of these subjects.

**Bringing Content In Stingray**

To bring our content to Stingray, we can use 2 methods. But each method is used in a specific context. Each time you export or send objects to Stingray, they become an entity called a Unit. So, you need to plan accordingly. If you intend to move, make visible or invisible or use multiple times, it’s important to group the objects accordingly.

**Environment and Ride_01 are 2 Different Units**
All static assets can be sent to Stingray directly with the Send command. That is true for all environment assets, props, and the roller coaster structures.

If something is animated, then, you need to go through FBX. When you export through FBX, the animation is baked. If you intend to export multiple animations to build a state machine of animations, you can use the Game Exporter. But that’s not our case for this project so we’ll use simple FBX export. That’s what we’ll use for the wagons and the camera.

Connecting the HMD to the Ride (in Flow)

Once we will have all our content in Stingray, we need to define where we want the HMD to be. If we want to be in the ride, we need to connect a few things so that the POV is right there in the wagons.

With the Oculus, all you have to do is set the active camera to be the animated camera from 3ds Max.
With the Vive, you need to set the tracking space to follow the position and the rotation of an animated object. Since the tracking space needs to be grounded, we need to connect it with an object that stands on the ground of the wagon. We can use a simple plane for that. As long as it is animated to follow the wagon.

![Animated Rectangle that Stands for the Vive Ground](image)

The tracking space can be positioned and oriented from a Flow node as well. We just need to connect the node to the transformations of the animated rectangle that stands as the floor.

![Connect the Tracking Space to the Animated Rectangle with the Set Tracking Space Node](image)

Finally, to play the animation, we'll use an animation play node and connect it to an input device of our choice.

To learn how to send content in Stingray and connect the nodes in Flow, watch the following video tutorial:

*07 Turn To VR.mp4*

Alternatives

One of the first use case of building roller coasters in VR with the ability to quickly create new ones and change the existing ones is the idea of ride alternatives. You want the user to try a few different rides and maybe pick the one they prefer.

For ride alternatives, we can assume that we have imported multiple rides with multiple animations from 3ds Max. Once they are all inside Stingray and from the same scene, we can pick the one rails and train we want to see and experience.
You can let the user choose which track by using an input from the keyboard or a button on a controller. From there, we need to hide everything that we don’t want to see. So, in our case, we would hide the tracks and the train. This is done by using the Set Unit Visibility node.

![Set Unit Visibility for Tracks and Trains](image)

Also, we need to show the track and train of interest by using the same node and set its status to true: visible. Based on the selected input, we need to set the correct active camera or attach the tracking space to the right object.

And, we can play the animation of the desired track. Once all of these are done, we can ride the track of interest. And if we attached tracks to numbers hotkeys, we can quickly switch tracks and show different rides. For approval process, this is great since the users can launch multiple experiences and choose the ones they like.

To see how this is done in Stingray, please watch the following video tutorial:

*08 Alternatives.mp4*
Animated Environment Assets

In an environment, it is always better to have some motion around. Especially in a VR experience where people can look all around. You need to give them something to see. And it will feel more “alive” if it moves. We don’t want to go into complex animation systems for environment assets. We want something that always moves and that you don’t have to trigger manually. You just want it to be there without having to worry about it. Here are a few ways to do this.

First, we can use regular animations coming from 3ds Max. They usually come in the form of PRS or skinned animations. Animations that will continuously play will benefit from being looped seamlessly. When you have such an animation, you can make it part of the asset by creating an animation controller and setting a default animation. This way, every time that this asset will be added in the scene, it will be animated by default.

Shaders also have animation tools in the form of specialized nodes. The most common ones are the Panner which makes textures moves long an axis. It’s animating the UV space of the shader.
Also, for plants and slightly moving objects like if they were in the wind, you can use the vegetation shader. It creates a wave like position offset that you can feed in the position offset of a shader. This is very helpful.

![Vegetation Node Produces Slight Vertex Movement to Simulate Wind](image1)

### Vegetation Node Produces Slight Vertex Movement to Simulate Wind

Stingray has a particle system as well and just this could be the subject of a separate class. Fire, explosions, smoke, mosquitos, you name it. Particles can be very creative.

![Particles in Stingray](image2)

### Particles in Stingray

Finally, Stingray knows about all dependencies when you create an asset. So, you can export using Stingray’s browser and it will package everything you need in a single file so that you can have the same intelligence when re-importing the asset using the browser. If you have an animated asset with a controller, it will all be packaged and when you bring it back in a different project, it will behave the same. I call those “smart assets”.

To see how to setup all these types of animations, please watch the following video tutorial:

*09 Animated Assets.mp4*
Class Files

All files used in this class can be found under this link:

https://www.dropbox.com/sh/dd6racpwkykzo10p/AACTibWfVI11ZIL_zWs0lXFNa?dl=0
Password: IWasThere

It includes the following:

- Distribution Files used in the Class
- The 3ds Max project
- All video Tutorials
- Final Ride Builds for HTC Vive and Oculus
- PPT Presentation and Handout for the class