

# Crowd System Walkthrough

## [1 Overview](#)

## [2 How the project works](#)

### [2.1 Import skeleton data](#)

### [2.2 Import restpose geo and joints](#)

### [2.3 Camera and render settings](#)

### [2.4 LoD selection](#)

### [2.5 Casting data and overrides](#)

### [2.6 Texture data and overrides](#)

### [2.7 Build crowd character geo](#)

#### [2.7.1 BodyType\\_OpScript](#)

#### [2.7.2 RootTransform\\_OpScript](#)

#### [2.7.3 LodSelectGroup](#)

#### [2.7.4 PruneParts\\_OpScript](#)

#### [2.7.5 CalcAnimBounds\\_OpScript](#)

#### [2.7.6 AnimProxy\\_OpScript](#)

#### [2.7.7 SkinEnvelope\\_OpScript](#)

#### [2.7.8 RigidBind\\_OpScript](#)

### [2.8 Clean up no longer needed hierarchy](#)

### [2.9 Resolve LookFiles](#)

### [2.10 Render with PRMan 19 and Render with Arnold](#)

## [3 Credits](#)

## 1 Overview

This example project is designed to show how a flexible pipeline for handling crowd data, including deforming geometry at render time, can be created in Katana. It is also designed to demonstrate two of the main new features in Katana 2.0: *OpScripts* and *Graph State Variables*.

The ingredients used in the scene are:

- Raw **skeleton data**, consisting of hierarchies of joints with animating transforms. This has been exported from a crowd simulation system, and is imported into Katana using Alembic files.
- A **kit of geometric parts** used to build the geometry for each character. There are a number of different parts such as 'body', 'top', 'bottom', 'hat', 'glasses' and 'shoes', with optional variants for each parts such as 'jeans' and 'khakis' for the trousers, or 'topHat' and 'cowboyHat' for the hats. There are also two different

level-of-detail sets of geometry for each set of parts, representing 'low' and 'high' levels of detail.

- A set of **texture variants** that can be used with each part.
- A **skeleton in a rest position** that matches the shape of the geometry in the kit of parts.

There is also metadata associated with these ingredients that is used in the construction process:

- Each piece of geometry in the kit of parts has metadata that describes whether the geometry is to be rigidly bound to a particular joint, or if it is to be soft-enveloped with a number of joints influencing the position of each vertex. For soft enveloping, the geometry includes shape weight data for the percentage influence of each joint on each vertex.
- Metadata that describes the options that are available for each part: the names of the different geometry variants, and the texture variants that are available for each piece of geometry.
- The rest pose skeleton also include bounds information for some joints, indicating the maximum bounds for any geometry directly influenced by this joint. These per-joint bounds are used to calculate animating bounds for each character.

This demo scene shows OpScripts being used to do a number of things that would have been more difficult with the older Scene Graph Generators and Attribute Modifier plug-ins. In particular, it shows operations that manipulate the hierarchy structure, such as copying sections of the hierarchy from one location to another, and removing parts of the hierarchy under script control. It also shows how operations such as deforming geometry and using the OpenEXR Imath library can be performed in OpScripts.

The OpScripts also show how multi-time sample data can be handled for motion blur. In particular, animation data from the skeleton joints is used to create appropriate multi-time sample data for joint transformations, animated bounds and the final deformed geometry.

The demo scene is also designed to show how Graph State Variables can be used to set up a single project that can be used for a number of shots, including handling multiple render outputs and customizing edits and overrides based on the context of which shot or output is being rendered.

## 2 How the project works

Here we describe the main functioning parts of the project, as shown by the division of nodes into different Backdrop areas.

The demo is set up to control 3 different shots in the same Katana project. This is done using a global Graph State Variable called 'shot' which sets the current shot number. In

the variables area at the top of the UI you can see the current value of 'shot', and you can change it to new values. Valid values for this project are '10', '20' and '30'.

A second Graph State Variable called 'region' is used to communicate which of the Render nodes is being rendered from. The idea is that different render nodes are designed to represent different regional variations of the production ('USA' and 'UK'), with some differences to the scene we want to render depending on the region (in particular, cowboy hats aren't allowed in the UK version). The 'region' graph state variable is used to control which edits and overrides are applied depending on which regional variation we're rendering.

The following tips may help you explore the project:

- You can jump to each named Backdrop area by using the 'J' keyboard shortcut and selecting the Backdrop you want to jump to.
- You can open up the scripts from OpScript nodes in whatever editor is specified in your **externalTools.editor** preference by clicking the **Edit in <editor>...** button above the script parameter. We suggest opening up and looking at the different OpScripts used in the project.
- The influence of Graph State Variables on which nodes contribute to the scene can be seen by enabling the **Dim Nodes Not Contributing to Viewed Node** option in the **Node Graph** tab's **Edit** menu.

## 2.1 Import skeleton data

The raw animating joint data for each character is read in by the *AlembicIn\_OpScript* node.

There is a different alembic file containing the animating skeleton data for each shot, contained in the 'massive' directory. The *AlembicIn\_OpScript* node uses the Graph State Variable called 'shot' to determine the path to the Alembic file to read. The actual reading of the file is executed using `Interface.ExecOp()` to call the AlembicIn Op.

If you view the scene from the *ViewSkeleton* node, expand the whole scene, and look through the main shot camera, you should be able to see a simple representation in the Viewer of the raw skeleton data. If you scratch up and down the timeline you can see the data animating.

The other nodes in this Backdrop are to do various utility things such as:

- Set the location type at each skeleton root to 'component' to make it easy to open the scene graph for each character without opening the whole skeleton.
- Calculating the global root transform and the inverse transform for each joint relative to the skeleton root, since these will be needed by downstream nodes. The calculation of these transforms is designed to correctly handle multi-time sample values if necessary for motion blur.

## 2.2 Import restpose geo and joints

The *KitOfParts* group contains the nodes used to read in the kit of geometry parts that will be used to build each crowd character, and create some additional meta data such as instanceID attributes on parts of rigid geometry so that they will be automatically instanced when rendered with Arnold or PRMan 19.0.

The *KitOfParts* also has nodes to help conveniently view the geometry. If you select the *KitOfParts* as the view node, expand the whole scene graph, and look through the camera in the Viewer you should see the different bits of geometry conveniently laid out.

The *RestposeJoints* group reads in the final main ingredient needed: a skeleton in a position that matches the shape of the geometry. This is used when calculating deformations. Additional meta data is also placed on the restpose skeleton, such root and inverse joint transforms that will be needed by downstream nodes, as well as bounds for some joints that will be used to calculate animating bounds for each character.

## 2.3 Camera and render settings

This Backdrop area contains separate *CameraCreate* and *GafferThree* nodes for each of the three shots. The *VariableSwitch* node uses the value of 'shot' to determine which of the three branches contributes to the downstream nodes.

By default motion blur is switched off. To see the effects of processing multi-time sample data, switch on motion blur by enabling the *MotionBlurSettings* node.

## 2.4 LoD selection

These nodes divide the characters in each shot into two different collections:

- *fgCharacters* - foreground characters that will be rendered with full soft bind skinning for deforming parts.
- *bgCharacters* - background characters that will be rendered with soft deforming parts replaced with rigid bound geometry that can make use of instancing to make rendering of a large number of characters more efficient.

The three *CalcZDistLodCategories* nodes are used to calculate the minimum z distance from the camera to the root location of each animating character over the shot range, and create the *fgCharacters* and *bgCharacters* collections using that data together with a range value that defines the division point between the collections.

The actual calculation of the minimum z distance from camera for each character is done using a Python script that executed by a button on the node. There are additional notes describing how the *CalcZDistLodCategories* nodes work in the project.

The collections are then used to set an attribute called 'character.lod.type' on the root location of each character that determines which LoD that character will use. Downstream those values are used by *LodSelect* nodes to remove geometry that isn't needed.

## 2.5 Casting data and overrides

The nodes in this area set metadata for each character that controls which geometry variants each character is to use.

The data is stored in attributes at the root location for each character, such as '/root/world/geo/characters/man\_1'. The convention used is to store the name of the variant to be used for a given part type as 'character.<partType>.type'. For instance, 'character.body.type' holds the body type to use, and 'character.hat.type' the hat type to use.

The *CharacterCastingData\_OpScript* assigns the casting of geometry parts for each character randomly based on the range of options available indicated at '/root/reference/restposeGeo/castingData'.

In a more realistic pipeline some (or all) of the casting data for characters may come in from other external sources, such as with the skeleton data from a crowd simulation system, or from an additional data file created by the layout department. A mixed pipeline could be set up where any explicit casting available from other sources in the pipeline is used first, and any character or part that doesn't have casting attributes set by those sources then gets automatic random casting using Katana nodes like this *OpScript*.

If you want to modify the casting data directly in the Katana project, custom edits and overrides to the casting data can be placed after the *CharacterCastingData\_OpScript*. Three example overrides are shown:

- *Man10\_PoloTop*: an *AttributeSet* node that sets man\_10 to use the polo top.
- *PruneMan2\_shot20*: a *VariableEnableGroup* that shows an override designed to prune man2 from shot 20.
- *CowboyHatsOverride\_UK*: A *VariableEnableGroup* that shows an override designed to only run if the 'UK' regional variant is set. It looks for any character with the 'cowboyHat' hat type, and changes the hat type to 'topHat'.

By enabling and disabling these nodes you should be able to see how they affect the final rendered scene.

## 2.6 Texture data and overrides

The nodes in this area set metadata for each character that controls which texture variant to use for each part, based on the casting metadata set in the nodes immediately above.

The *CharacterTextureData\_OpScript* node looks at attribute data at the root of each character that declares which geometry variant to use for each part, then gets the list of

texture variants that are available for that part from `/root/world/reference/restposeGeo/castingData`, and picks one of those at random. The value is then stored at the root of the character as attributes of the form `'character.<partType>.texture'`, for instance `'character.hat.texture'` holds which texture variant to use for a given character's hat.

Immediately downstream of the *CharacterTextureData\_OpScript* node is the area to place custom edits and overrides for texture casting. An example override is shown using an *AttributeSet* node to change `man_18` to use `textureA` for the top.

## 2.7 Build crowd character geo

This is the section that does the heavy lifting, copying the correct body variant kit of parts to each character, pruning out the parts that aren't needed, and applying soft or rigid binding geometry to the animating joints.

### 2.7.1 *BodyType\_OpScript*

This *OpScript* makes a copy of the hierarchy of geometry parts for the correct body type from `/root/reference/restposeGeo` to each character.

The script runs at the root location of each character, and uses the value stored in `'character.body.type'` to calculate the path for the required body parts in the `restposeGeo`.

The two Lua statements that do the heavy-lifting to make a local copy of the hierarchy are:

```
Interface.CopyLocationToChild("geo", bodySource)
Interface.StopChildTraversal()
```

The first function call creates a child of the current location called `'geo'` and makes a copy of all locations from path indicated by `bodySource` under it. The second stops this same operation also running at any child locations, which would result in additional copies of the hierarchy under each child location.

One thing to note when copying locations: attributes are held as references to the attribute at the original location until they are modified. This means that copying the entire kit of parts to each character is a light operation that is reasonable to do even for a large crowd of characters.

### 2.7.2 *RootTransform\_OpScript*

This *OpScript* copies the root transform for each character from the root of the skeleton to the root of the geometry.

### 2.7.3 *LodSelectGroup*

This group node contains two conventional *LodSelect* nodes that retain the correct geometry from the kit of parts based on the value of `'character.lod.type'`. One thing to note

is that the *LodSelect* nodes use CEL statements that access the inherited value of 'character.lod.type' to make sure that the appropriate *LodSelect* node runs depending on the value of that attribute set at the root of each character.

#### **2.7.4 PruneParts\_OpScript**

This *OpScript* prunes out the geometry that isn't needed from the kit of parts for each character based on the casting data.

The script works in the following manner:

1. The script run at every location of type 'part-type'. This is a custom location type that has been used to mark the root location for every optional part in the rest pose kit of parts. For instance: 'hats/baseballCap', 'hats/cowboyHat' and 'hats/topHat' are all set to be locations of type 'part-type'.
2. The script then finds the name of the part type (such as 'hat') by looking for the inherited value of the attribute 'character.partType'.
3. The script then compares the name of the current location (such as 'topHat') with the value set in the character's casting data for which variant to use. If they don't match it deletes the location.

#### **2.7.5 CalcAnimBounds\_OpScript**

This script calculates an animating value for the bounds for each character based on the animating joint positions. Some of the joints in '/root/reference/restposeJoints' have bounds to indicate the maximum bounds for any geometry directly affected by that joint. The script uses lmath to calculate the minimum size of a bounding box that would contain the bounding boxes for all the individual joints.

If the transforms for the joint have multiple time samples then separate bounds are calculated for each time sample.

#### **2.7.6 AnimProxy\_OpScript**

This *OpScript* shows how proxy geometry to be seen in the Viewer can be procedurally created.

The script calculates an animating proxy to be used for each character. In this case we simply use the joint bounds previously used to calculate the bounds to also be viewable as geometry.

The script works by calculating the values of the geometry attributes that would be needed by a polygon mesh for the proxy.

The details about how this is then turned into a proxy to only be shown in the Viewer are a little more complex, but the code in this *OpScript* can be considered to be 'boilerplate' code for how such an *OpScript* could be used for other geometry data. What the script is actually doing is passing the calculated geometry attribute values as arguments to a

deferred *OpScript* defined under 'proxies.viewer'. This is the convention for declaring a deferred Op to only run in the Viewer. This Op creates a child location called 'proxyGeo', sets the geometry values that have been passed to it, and sets the location type to 'polymesh'.

If the hierarchy for a character is expanded to the 'calcBounds' location ('man\_XXX/geo/calcBounds'), you should be able to see the animated proxy in the viewer. This location is set as type 'component' to make it easy for the users to expand to the level that shows the proxies. For instance, if the scene graph is fully collapsed (so that only '/root' is shown in the **Scene Graph** tab), if you double click on '/root' the scene graph will be expanded to show the component locations, which will show the proxies.

### **2.7.7 SkinEnvelope\_OpScript**

This script does the heavy lifting to calculate the deformation of soft enveloped geometry.

In the kit of parts any piece of geometry that needs to be soft enveloped has arbitrary attribute values that represent skin weighting information. This data can be held in array attributes of the form 'geometry.arbitrary.joint\_\_<joint name>' where '<joint name>' is the name of one of the joints that affects this piece of geometry. The values are the percentage weight that the joint affects each vertex.

This data was authored and exported from Maya, using a Python script that extracted soft enveloping weight data and converted into a form suitable for export as Alembic arbitrary attribute data.

The script finds each piece of geometry that needs to be soft deformed, then loops through the list of joints that affect the geometry, and uses lmath to calculate the new deformed position of each vertex.

If the transforms for the joints have multiple time samples then a separate set of deformed vertex positions are calculated for each time sample.

This script should be seen as an example of the type of geometry processing that can be done using *OpScript* nodes. Other similar geometry deforming operations, such as a wrap deformer or a relaxation deformer such as Delta Mush, could be similarly implemented.

For more efficiency it would probably be better to implement a process like this as a native C++ Op, but *OpScripts* provide a convenient way to prototype functionality that could then be re-implemented if necessary as an Op, and in many cases may have sufficiently high performance.

### **2.7.8 RigidBind\_OpScript**

The final node involved in processing the geometry for each character processes geometry that needs to be rigid bound.



As well as soft bind data, the geometry in the kit of parts also contains arbitrary attribute values describing rigid binding. This is simply an attribute called 'geometry.arbitrary.rigidBind' that gives the name of the joint this piece of geometry is to be bound to.

The script finds whether a piece of geometry should be rigid bound, find the transform for the relevant joint, and applies it to the geometry.

If the transforms for the joint has multiple time samples then a separate transform is calculated for each time sample.

## **2.8 Clean up no longer needed hierarchy**

This section prunes out some parts of the hierarchy that are no longer needed now that the geometry for each character has been built. This is simply good housekeeping: the scene will render fine without doing this, but it makes the final hierarchy cleaner.

## **2.9 Resolve LookFiles**

Look files containing look development data, in the form of materials and material assignments, are assigned to each character and resolved.

## **2.10 Render with PRMan 19 and Render with Arnold**

These areas are for render specific nodes for PRMan and Arnold, and final Render nodes. Each section has separate render output nodes for 'USA' and 'UK' versions. The *VariableSet* nodes just above the Render nodes set values for a graph state variable named 'region' which can be used to create context specific edits and overrides upstream, so that the scene data gets processed differently depending whether we are rendering the 'USA' or 'UK' version of the shot.

# **3 Credits**

- Skeleton animation data created using Massive
- Geometry and textures created using Mixamo