MOANA ISLAND SCENE

Heather Pritchett & Rasmus Tamstorf

Introduction

This data set contains everything necessary to render a version of the Motunui island featured in the 2016 film “Moana” from Walt Disney Animation Studios (WDAS). The hope is that this will facilitate new research in computer graphics including improvements to algorithms related to rendering. However, other types of research are also encouraged. The scene is chosen to represent some of the challenges we currently encounter in a typical production environment. Most notably this includes large amounts of geometry created through instancing as well as complex volumetric light transport. There are many other challenges which are frequently encountered in production and which are not represented in this scene (examples include motion blur and a large number of light sources to name just two). Therefore, there is no tacit implication that this is *the* most challenging scene by any metric. Still, we sincerely hope that this will be a useful dataset for developing, testing and benchmarking new technical ideas.

License

Contents

The scene is made up of 20 elements containing meshes with more than 90 million unique quads and triangles along with 5 million curves. Many of the primitives are instanced many times giving rise to a total of more than 28 million instances of everything from leaves and bushes to debris and rocks. When everything is fully instantiated the scene contains more than 15 billion primitives. The data for all of this is organized as follows :

island/ - Base directory. All paths are relative to this directory.
island/README.pdf - This file.
island/License.txt - The license for this data set.

island/json - This directory contains json files that include all the data needed to define the lights, cameras and elements in the scene. Each element has its own subdirectory that includes both a main file with all instances and transforms and a separate file with the materials associated with that element. The json-based material files include the full brdf definition as well as the original bindings.

island/obj - This directory contains the object files which define the geometry for each element. For each element there is a subdirectory which contains a standard .obj file for the object, a corresponding material binding .mtl file, and a json-based .hier file. More information on the usage of these files can be found below under “Details and caveats”.

island/animation - Per frame data for the ocean and the shot camera which are the only animated elements.

island/ref - Reference images from our simplified production scene file using both Hyperion and pbrt. Also contains some screen dumps from Maya of the different elements.

island/textures - All the textures used by the elements. These are stored in Ptex format.

island/pbrt - Scene files for pbrt-v3. This is an example of how the data in the above directories can be used. These files can be regenerated using the pbrt scripts (see below).

island/scripts/pbrt - The python scripts used to create the pbrt scene files from the raw inputs above. These are provided as an example of how to access the data.

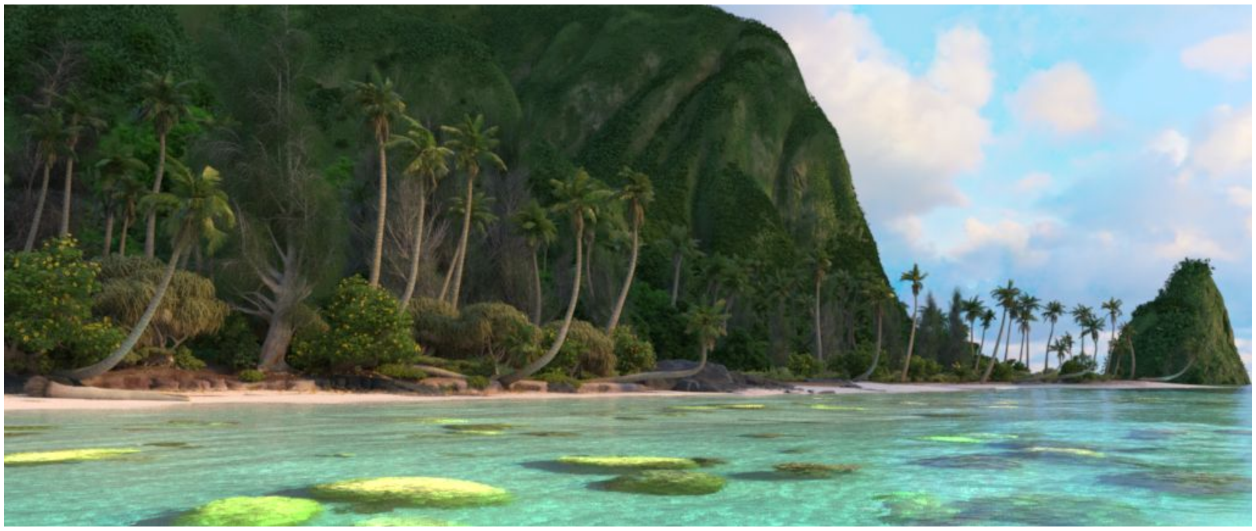
island/scripts/embree - A modified version of the pbrt scripts used to convert the raw input into the XML format used by the sample path tracer that comes with Intel’s Embree.

All the data is provided in three tar-files. The base package provides everything to necessary render a single frame. The animation package adds everything from island/animation, and the pbrt package adds the pbrt scene files.

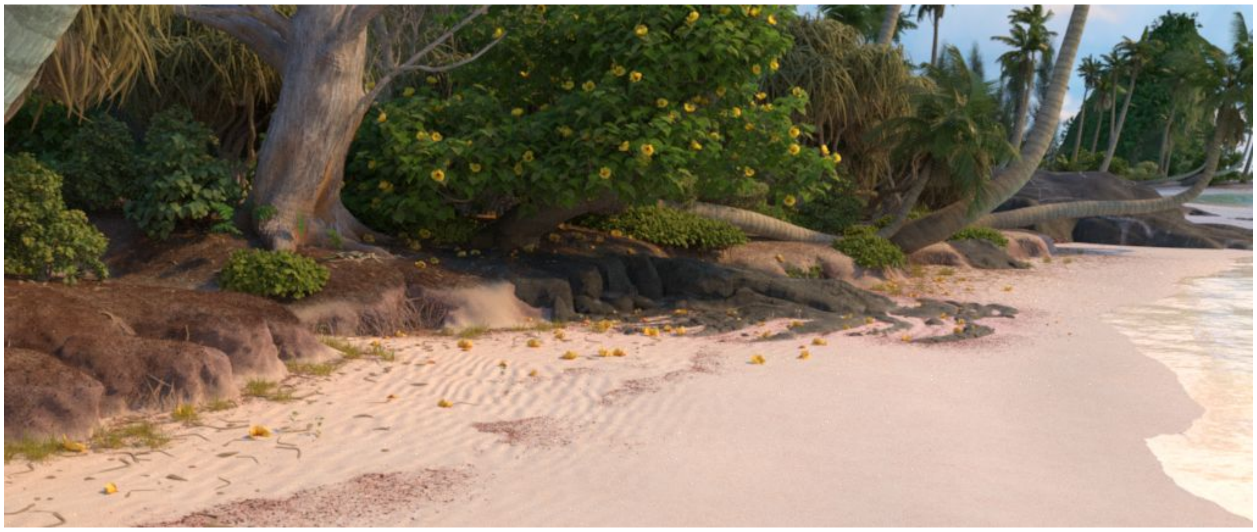
In the following, we first show a number of different camera views of the scene. Then we define our terminology and discuss some of the details related to the process of creating the data set. We also point out some limitations in our PBRT version of the scene, and we outline potential areas of future work. In the appendices we give an overview of all the elements in the scene, and we cover our data formats in detail.

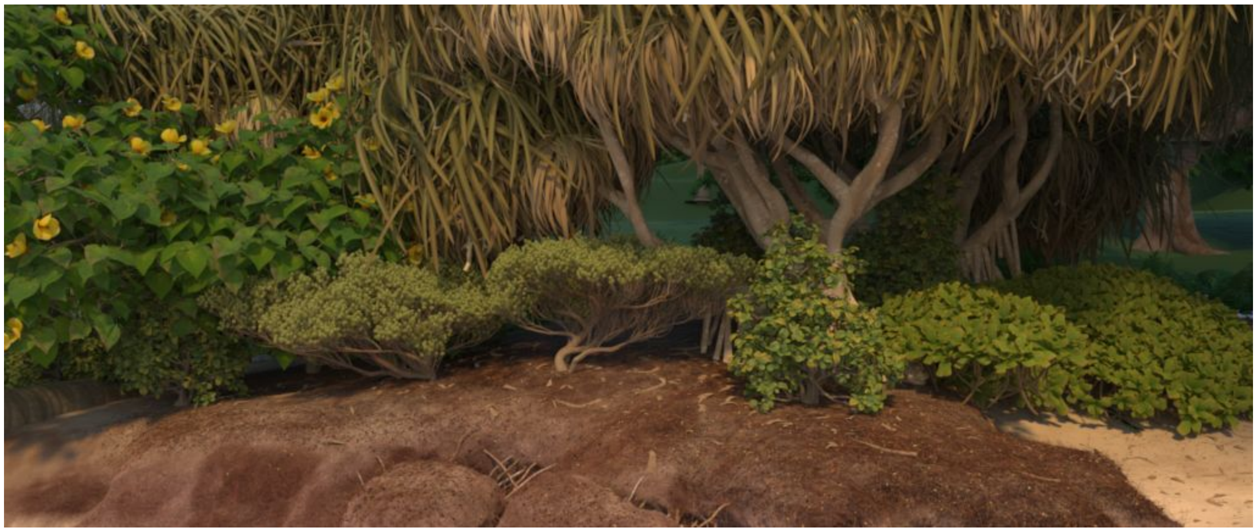
Cameras

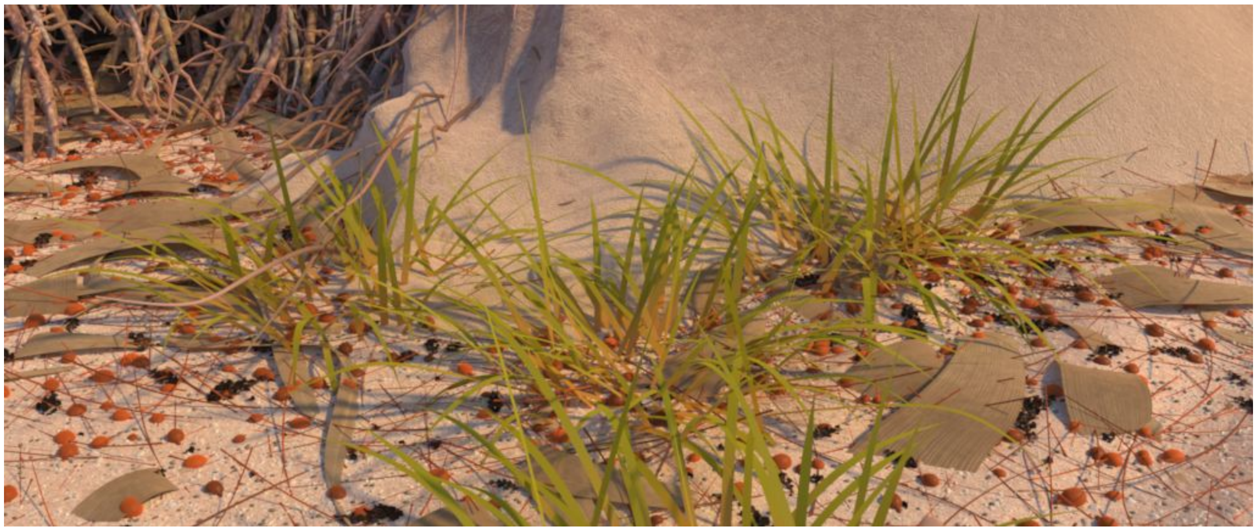
A total of seven cameras are included with this shot. The first of these is the primary “shotCam”, which was used for the original animated test, while the remaining 6 cameras provide various other views of the scene. The following images rendered in Hyperion show the view from each camera.



[shotCam]


[beachCam]


[dunesACam]


[grassCam]

[birdseyeCam]

[palmsCam]

[rootsCam]

Terminology

To describe the data set in more details, we have tried to use words and references that are common to the computer graphics community and that anyone with a basic familiarity with physically-based rendering will know. However, there are some terms that might not be familiar to those without a production background and some terms may even be specific to WDAS. We are going to highlight those here and provide their meaning for the purposes of this project.

Scene vs shot - For all intents and purposes, “the scene” and “the shot” are the same thing and used interchangeably throughout this document. The scene or shot is the collection of all data used to generate the final image, including geometry, materials, textures, lights and camera.
Element - An element is a single unit representing a thing in the shot. An element will have geometry, materials and textures associated with it.
Instance - An instance is a copy of an element. All instances share the same materials and textures. Most instances share the same geometry, but not all. In cases where the element has been animated to show motion, or deformed to add diversity to the layout, a new geometry file is needed.
Variant - A variant is an instance with a unique name. It is referred to from another element as part of a primitive description.
Primitive description - This is a collection of transform data and geometry references used to create a distribution of *something* over some piece of geometry. Think of fur on an animal, grass on a field or coral on a seabed. This is sometimes referred to merely as a **‘description’**. For the sake of this document, descriptions distribute two different types of data: curves or archives.
Archive - This is a single unit representing a group of meshes that is being distributed along an element. It can represent something as small as a pebble or something as large as a tree. An archive has geometry, but usually shares materials and textures from its parent element. The exception to this rule is when primitive descriptions distribute other elements as archives. This is described in more detail below.
Ptex - The open source texture format used at WDAS. See http://ptex.us/. An important point to note is that this is a face-based format, so changes to the face count or the face order of a surface will break its texture.
XGen - Our in-house arbitrary primitive generator. All primitive descriptions in this shot were originally authored in Disney's XGen. Please note this is significantly different to the XGen included with Autodesk’s Maya.
Bonsai - Our in-house tool for procedural generation of vegetation. It was used to generate the majority of the tree shapes in the shot.
Hyperion - Our in-house path-based ray-tracer. See https://www.disneyanimation.com/technology/innovations/hyperion

What is and isn’t in the data ?

This scene was originally a production shot. As part of the process of making it possible to render it outside of WDAS, considerable changes have been made to both the data and the shot. In the following we describe how the new shot differs from our original.

Principled materials

All materials are provided in the Disney “principled” BRDF/BSDF form. In addition, we also provide a stand-in color for very simple shading. More information on BRDF/BSDF formats can be found here:

- https://disney-animation.s3.amazonaws.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf
- http://blog.selfshadow.com/publications/s2015-shading-course/burley/s2015_pbs_disney_bsdf_notes.pdf

In our production workflow, materials are often quite complex, with layers and masks that allow a single material to represent numerous variations across a single surface and be easily art-directable. For this project, those materials were simplified as much as possible. We baked down all our color and displacement layers into a single texture. Unfortunately, there wasn’t an easy mechanism to do the same for our “principled” shader controls. We opted to provide a single layer for each material, but in doing so we have lost small physical details, like the the translucency of a leaf at its edge.

Geometry and texture format

All geometry is represented in the Wavefront OBJ file format, which was chosen for its widespread use. All surfaces are Catmull-Clark subdivision surfaces, so each OBJ file contains a quad-mesh representing the control cage for the subdivision surface. We do not use any creases. We generally interact with subdivided meshes at a level that is representative of the authored shape, and, at render time, subdivide further.

All textures are in Ptex format and our internal version is identical to the Open Source version provided on github. For more information:

- http://ptex.us/
- http://ptex.us/ptexpaper.html
- http://github.com/wdas/ptex

The Ptex format is a per-face format, so it is tied to the geometry. Keeping the quad representation in the OBJ files means that we didn’t have to update the matching Ptex files. However, it also means that if you convert to triangles then you have to track the original face indices in Ptex and keep track of the quad orientation so you're mapping each triangle to the correct half of the corresponding Ptex quad.

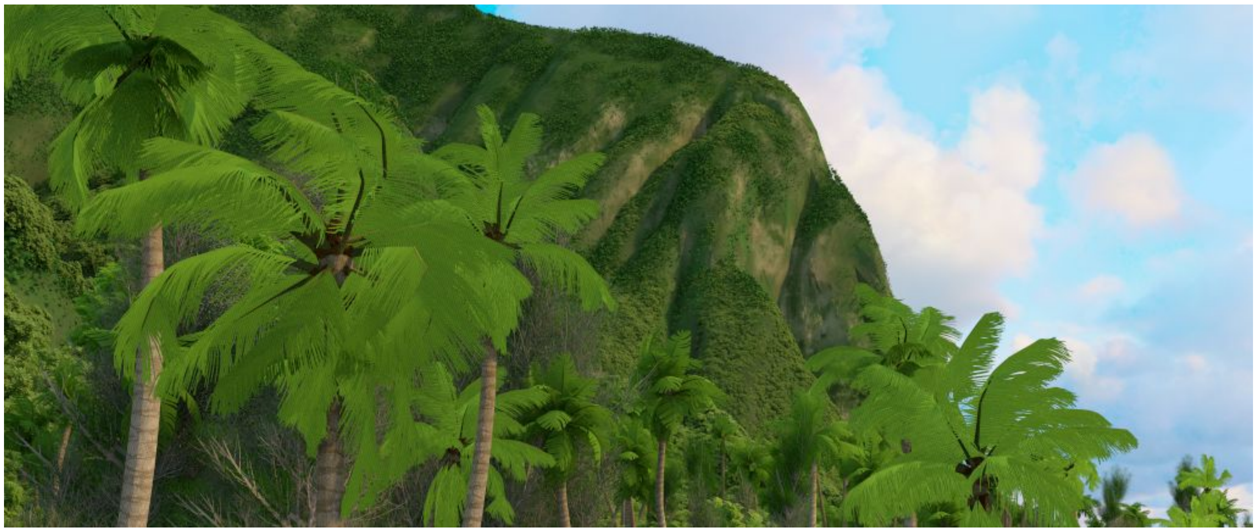
Color spaces

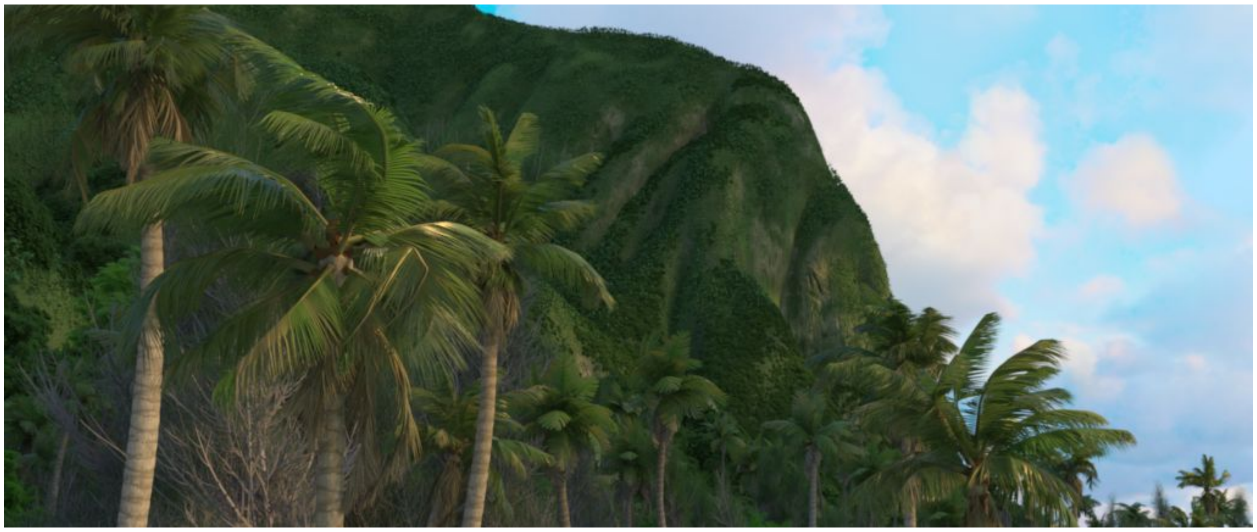
At WDAS, all artists work in a monitor-based color space that is visually linear. All of our colors, light, material and texture-based inputs are in this space. However, many renderers (including pbrt) expect all colors in true linear space. The easiest way to convert the colors associated with the lights and materials is to apply a gamma correction with gamma = 2.2.

It's also important to note that the Ptex format itself is color agnostic. For the textures provided in this dataset all colors are stored in the same monitor-based color space as the material parameters.

Curves

We make use of curves to represent grass, roots, and even a ribbon-like tree known as a pandanus. Our curves are generated in XGen using a format similar to what Maya uses. Consequently, our bake process generates mel files containing data for uniform B-splines. To ensure that the curve interpolates the first and last control points Maya will duplicate these control vertices and any user of this data will have to do the same. All of the curves are of '-degree' two, but vary in length depending on the complexity of the object. Unfortunately, our current export process simplifies the primitives associated with each curve considerably. In XGen, we have controls for the orientation, bend, tip and root width taper and even the color along the length of the curve. In addition, we can control all colors with maps based on the growth surface geometry. The fronds on the palm trees serve as a good example for this. Their color is based on a texture map painted on the frond stem (which is the growth surface) and changes along the length of the curve. In addition, there are complex expressions driving the width and orientation. While we can capture some of this data in a B-spline curve, we still lose a lot in visual variety as illustrated by the two palm trees compared below. *Note: See the "Our lights != PBRT lights" section below for some details on why the foreground palms appear brighter.*

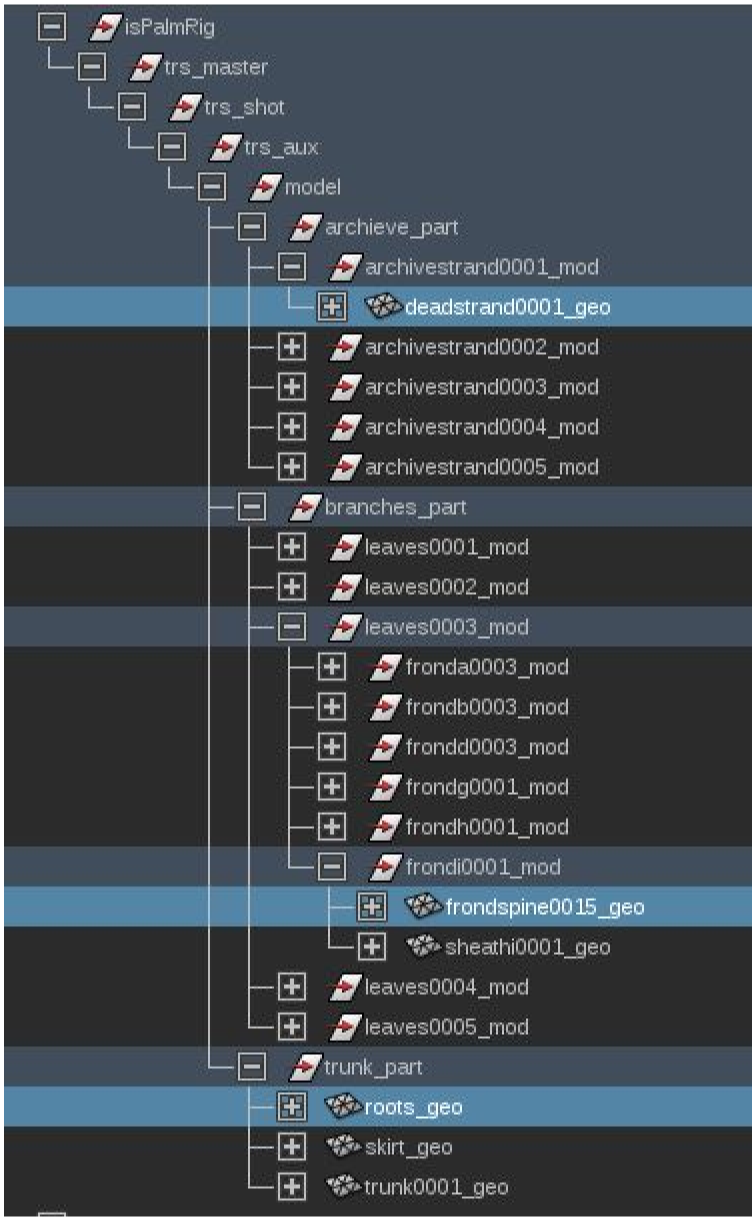

[PBRT render of exported data]


[Hyperion render of original data]

OBJ format and material binding

Using the OBJ format provides some challenges in properly representing our material bindings, since it doesn’t support hierarchical file names easily, nor does its associated material format understand our principled materials.

Our material bindings are wildcase based and utilize our production hierarchy. As an example, the following snapshot from Maya shows the ‘isPalmRig’ standard hierarchy :



```
isPalmRig
└─trs_master
  └─trs_shot
    └─trs_aux
      └─model
        ├─archieve_part
        │ ├─archivestrand0001_mod
        │ │ └─deadstrand0001_geo
        │ ├─archivestrand0002_mod
        │ ├─archivestrand0003_mod
        │ ├─archivestrand0004_mod
        │ └─archivestrand0005_mod
        ├─branches_part
        │ ├─leaves0001_mod
        │ ├─leaves0002_mod
        │ ├─leaves0003_mod
        │ │ ├─fronda0003_mod
        │ │ ├─frondb0003_mod
        │ │ ├─frondd0003_mod
        │ │ ├─frondg0001_mod
        │ │ ├─frondh0001_mod
        │ │ └─frondi0001_mod
        │ │   ├─frondspine0015_geo
        │ │   └─sheathi0001_geo
        │ ├─leaves0004_mod
        │ └─leaves0005_mod
        └─trunk_part
          ├─roots_geo
          ├─skirt_geo
          └─trunk0001_geo
```

Depending on the needs of the artist, he or she can bind the individual meshes (which generally end in _geo) to any group in the hierarchy. For example, binding to 'archive_part' would allow them to use the same material on all the dead strand pieces.

In order to provide some flexibility we keep the original wild-card bindings in the materials.json provided for each element. For those files to work correctly, the original geometry hierarchy has to be restored from the .hier file, which contains the original hierarchy for the element. Many of the bindings we use in production, depend on our original Maya-based groupings, which are stripped out during the OBJ conversion. We have therefore run a post-process to re-apply the bindings to the full hierarchy name of each mesh, and we have also inserted the material into the OBJ file as a 'usemtl' statement.

Sample hierarchy lines from the ‘isPalmRig.hier’ file:

```
"deadstrand0001_geo": "|isPalmRig|trs_master|trs_shot|trs_aux|model|archieve_part|archivestrand0001_mod",
"frondspine0015_geo": "|isPalmRig|trs_master|trs_shot|trs_aux|model|branches_part|leaves0003_mod|frondi0001_mod",
"roots_geo": "|isPalmRig|trs_master|trs_shot|trs_aux|model|trunk_part",
```

Our instanced geometry adds another challenge to material bindings. In our internal pipeline, an artist can bind a material to a primitive description, or to individual archives or curves that are being instanced. For example, the isBeach element has two primitive descriptions with different binding styles, xgStone and xgBonBabyGardenia.

The primary material used to shade xgStone is ‘archiveVolcanicRock’ and it is bound directly to ‘xgStone’. The xgStone primitive description consists of similar rocks and only needs a single stone material.

The materials used to shade xgBonBabyGardenia are ‘xgBonBabyGardeniaBranch’ and ‘xgBonBabyGardenia_Leaves’. The artist requires two different materials for the final gardenia plant that will be instanced by this description. They are bound to, respectively, ‘bonBabyGardenia_hTrunk’, ‘bonBabyGardenia_hBranch’ and ‘bonBabyGardenia_Instance’. These are groups within the geometry XGen creates.

The OBJ files can only contain the bindings as the final mesh names, so we cannot use the primitive description names like ‘xgStone’ or group names from the final instanced geometry like ‘bonBabyGardenia_hTrunk’ in the OBJ file.

If you look at the file json/isBeach/isBeach_xgStones.json, you can see that it includes certain OBJ files:

```
% grep obj/isBeach json/isBeach/isBeach_xgStones.json
    "obj/isBeach/archives/xgStones_archiveRock0001_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0004_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0006_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0005_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0007_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0003_geo.obj": {
    "obj/isBeach/archives/xgStones_archiveRock0002_geo.obj": {
```

These are the baked archive files we created to represent the primitive descriptions. Looking at a sample file, you can find the OBJ ‘usemtl’ statement that defines the binding:

```
[...]
g archiveRock0001_geo
usemtl archiveVolcanicRock
[...]
```

This is the per-mesh binding which we inserted into each OBJ file. You can verify that the material was correctly inserted by checking the values in isBeach's material palette: json/isBeach/materials.json.

```
    "archiveVolcanicRock": {
[...]
        "assignment": [
            "xgStones"
        ],
```

This works if you plan to use the bindings in the OBJ file. However, if you are working with only the geometry and need to generate your own bindings for the primitive descriptions, then you need more context than the raw geometry in the OBJ file. You will need to apply the hierarchy file for them to work, since that knowledge is not in the OBJ file. Nor does the OBJ file know about the name of the primitive description. In the 'xgStone' example above, the archive files for the rocks know their own hierarchy, but not the hierarchy of the beach. To get around this gap, when we check bindings for archives, we also check against the name of the file, which has the primitive description name, i. e. xgStones_archiveRock0001_geo.obj.

No compositing

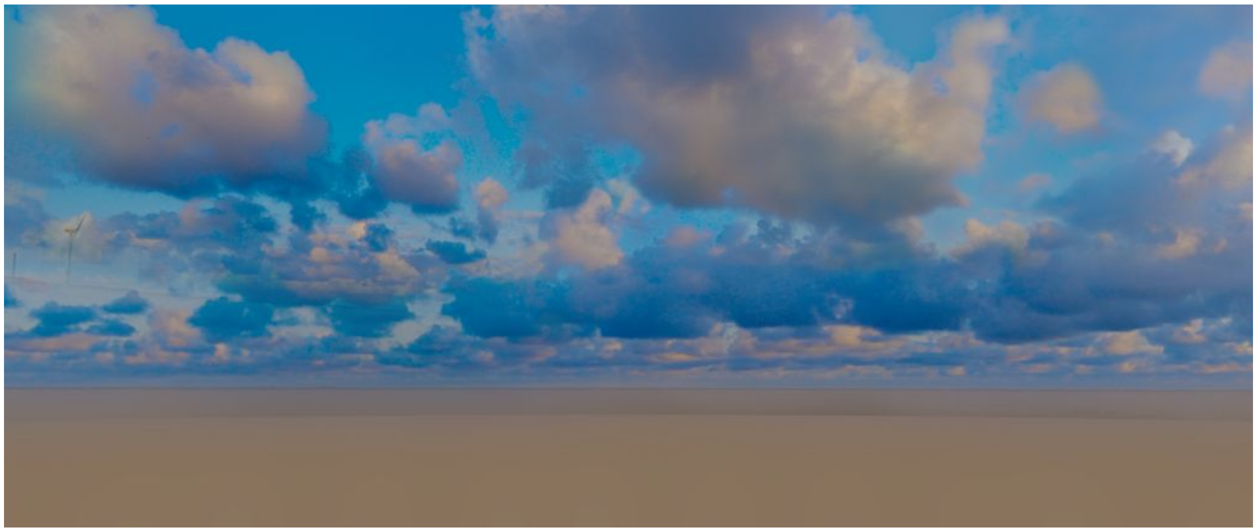
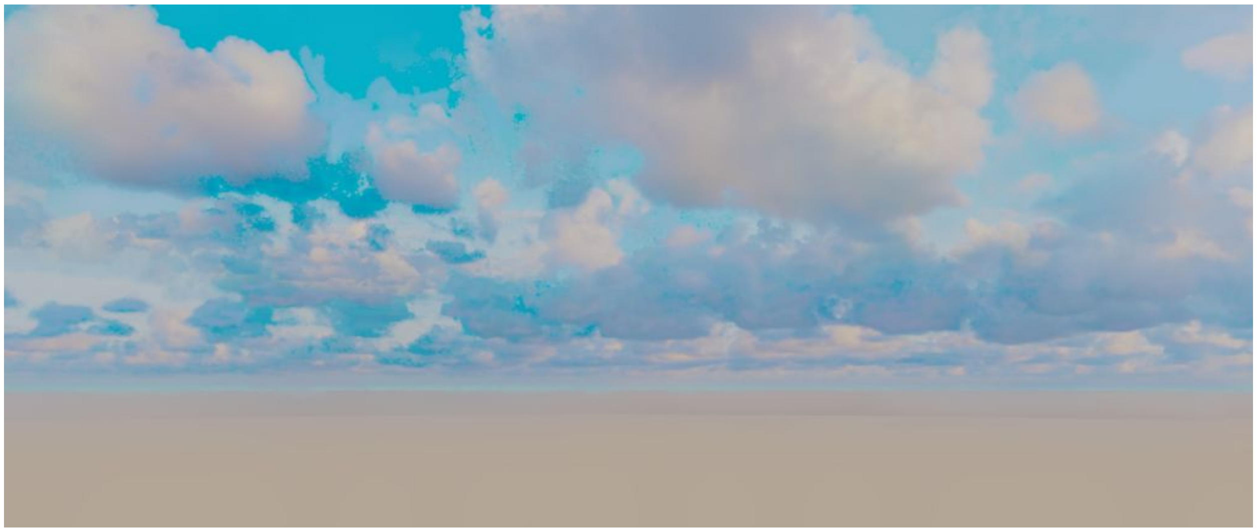
Our final production images are not rendered in a single pass. Lighting and stereo artists will break the shot up into different render passes, separating out background, midground and foreground elements at a minimum. They will also break out specific components and lights and recombine everything together with tweaks and adjustments to produce the final rendered images. For the island test, we removed all the separate passes and adjusted the shot to render as one pass. While this simplifies rendering, we lose some of the art direction that drove the shot in production.

Art-directed lighting

In the original shot, there were more lights than just the sun. In addition to the map-driven environmental lighting (see below) there were numerous spot lights and quad lights. These additional lights were designed to art direct the lighting in certain areas. As an example, there is a series of lights brightening up the beach as well as lights creating additional bounce lighting from the ocean up onto water-front foliage. We left most of these lights in the shot, however, their final contribution isn't always needed and, in most cases, should only apply to certain elements.

Skydome and environment lighting

There are two lat-long images associated with our infinite light source. In order to separate art direction of the background from lighting of the scene, we use one map for environmental lighting, and a second visible map for the radiance value of rays that don’t hit anything in the scene.



[textures/islandsun.exr (top) vs textures/islandsunVIS.png (bottom)]

Instanced geometry as elements

Much of the foliage on the island is the same primitive copied over and over again. In some cases, for large “hero” foliage, the element is a tree or a bush and numerous instances of that element are copied. For example, there are 13 copies of isGardeniaA. In the json file provided for isGardniaA, we provide 13 additional transformations that tells you how to place these instances.

[isGardeniaA and copies (Hyperion)]

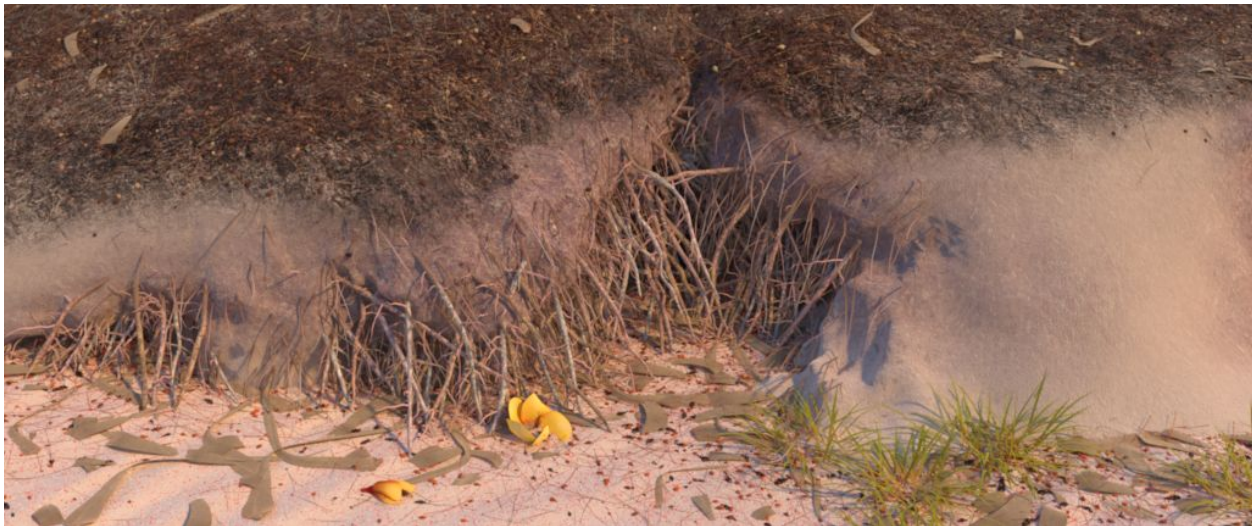
Instanced geometry as primitives

We create non-hero foliage directly on the underlying geometry (like isBeach) using either Disney's XGen or Bonsai to “grow” foliage everywhere we need it. For this project we provide OBJ files for all individual primitives and individual transform data for each instance, essentially creating a simplified reference format. However, this did not work in a few cases:

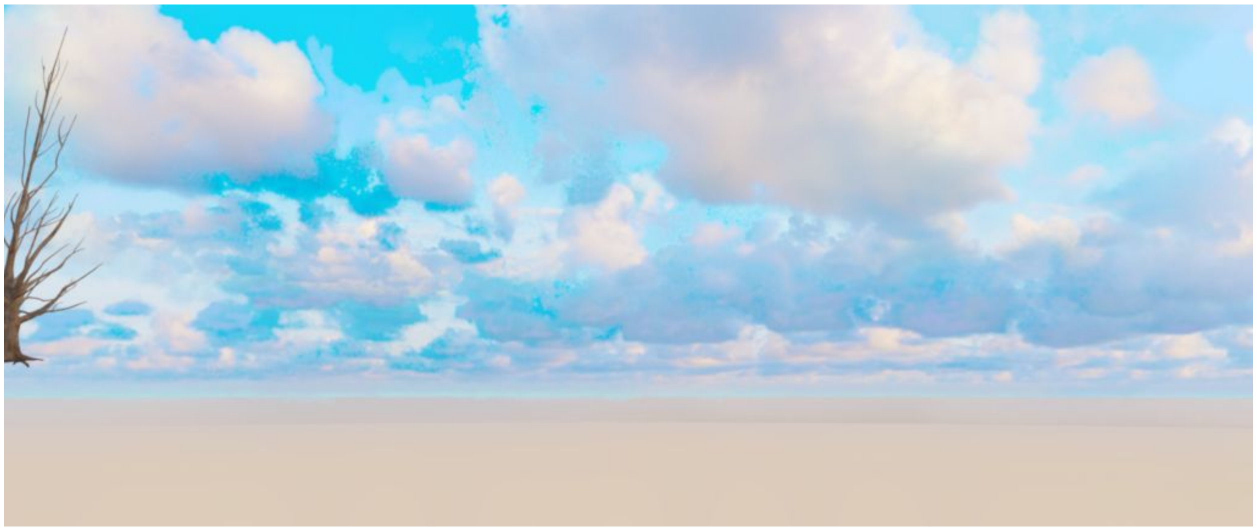
1. The Bonsai format procedurally generates the branch structures and this is not easily presented in a simple object/translation format. For those cases, we have baked down the Bonsai elements as much as possible.
2. The palm tree elements, despite being copies of each other, are rigged into different positions, so these require distinct baked geometry files per instance.
3. The coral and naupaka elements are composed of multiple “clumps”. These are not animated, but, like the palm trees, require distinct copies of the geometry to represent the different arrangement of the “clumps”.

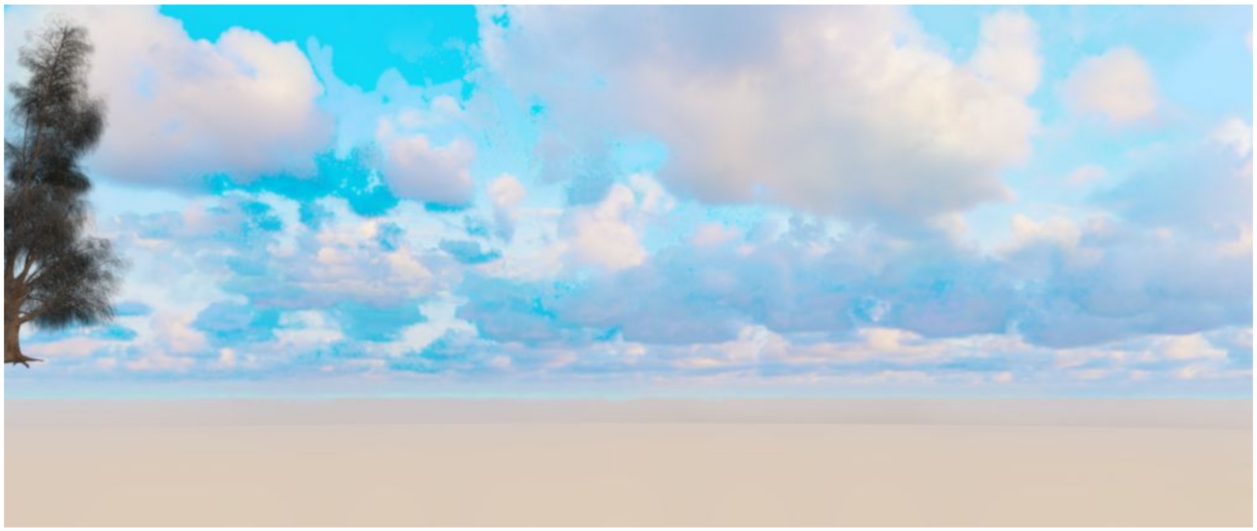
Instance types

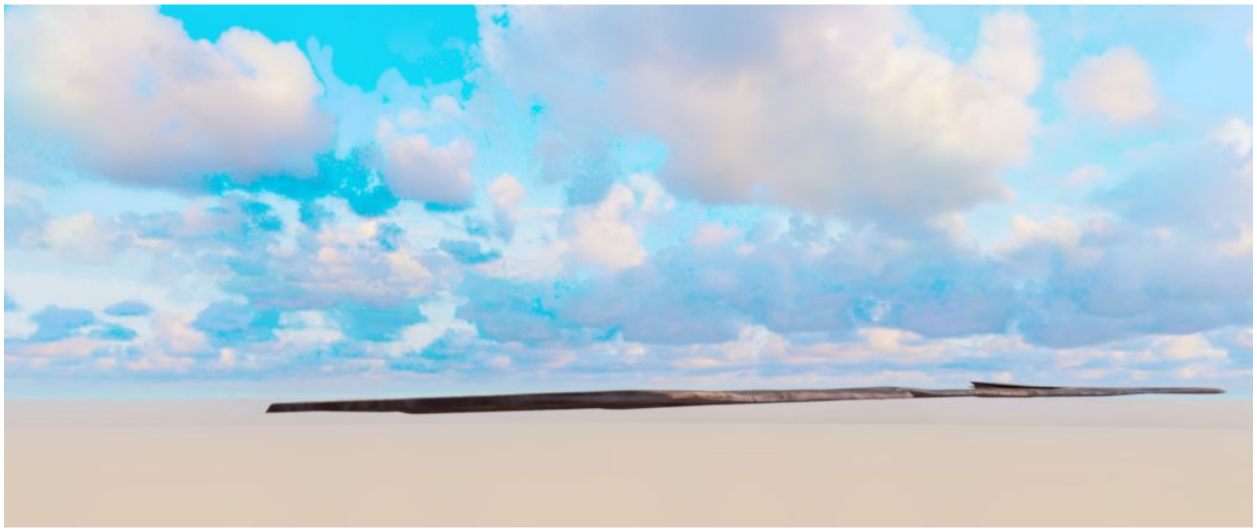
Disney's XGen is a powerful tool that provides numerous controls that can be used by the primitive description to distribute curves and geometry data. In the json file for each element, the curves and geometry data are distinguished using a ‘type’ property. Grass and roots might both be represented as ‘curve’ types, whereas the debris on the beach (which is of type ‘archive’) is numerous small geometry files of shells, stones, twigs, palm leaves and hibiscus flowers.

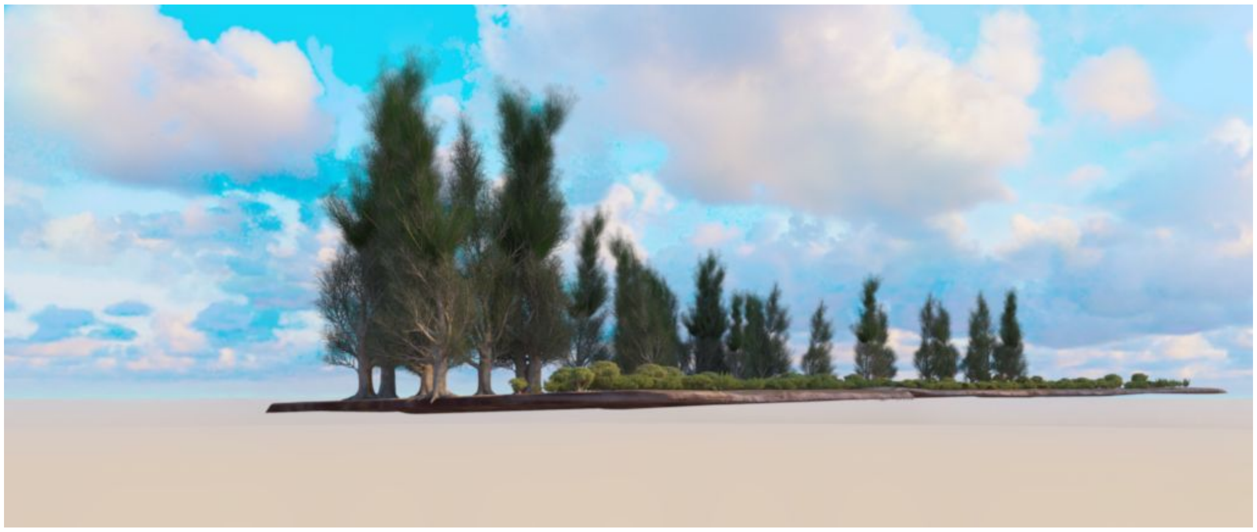

[closeup of isBeach and isDunesA showing primitive descriptions using archives and curves (Hyperion)]

Disney's XGen is also a wrapper for Bonsai. In internal scenes, Bonsai can be integrated in two different ways. In some cases, each XGen description represents a single tree, like isIronwoodA1 or isBayCedarA7. But it can also be treated as a distribution entity wherein a single archive description, like isDunesB/xgTreeSkyLine, creates several copies of isIronwoodA1.


[The element isIronwoodA1 rendered in hyperion without Disney's XGen. This is the bare trunk that the bonsai generates off of.]


[The full element isIronwoodA1. It consistents of a trunk + Disney's XGen primitive description that utilizes bonsai to create numerous branches, needles and seedpods in a hierarchical structure]


[The element isDunesB without Disney's XGen.]


[The element isDunesB is comprised of a ground plane and 5 Disney's XGen primitive descriptions. When bonsai is distributed through XGen as a distinct entity with multiple occurences, the trunk is considered part of the primitive and not the main element.]

The isDunesB has numerous references to other elements, including copies and lower resolution versions of isIronwoodA, isPandanus and isBayCedarA. We need a way to handle this reuse, and more specifically a way to represent elements that are not simply “instancing things” but are “instancing things-that-are-also-instancing-things”. Most of the elements in this category are elements that instance lower resolution variants and which can be baked down into a single file without too much of an impact. The remaining challenge is a specific primitive representation in isDunesB, xgTreeFill, which references several different variations of isBayCedarA. We have created a third type of instance, called “element”, which signifies that this instance has its own instances. This is paired with a new property in the main isBayCedarA.json file, called “variants”, that defines both the geometry and primitive file associated with each variant. For example, isBayCedarA has a variant “bonsai A” that is referenced in isDunesB’s xgTreeFill. For more information, see Appendix 2.

The ocean

The original shot was created using an early version of our ocean water pipeline. To be more representative of what was used in the movie, we upgraded the entire shot to the latest techniques used in the movie. We then adjusted and art directed the settings to match the original “minimalist” water as much as possible. The ocean in our updated test shot is a level set. We baked each frame directly out of Hyperion, captured both the high-resolution in-frustum portion and lower resolution outside portion. Unlike all the rest of the geometry, the ocean is triangulated as that is the final format Hyperion accesses.

The final look for film shots with the ocean involved two different materials. A fairly simple principled material applied to the surface itself, and then a custom “deep water” volume shader that could be fine-tuned to return the art-directed colors of the ocean depths (or shallows).

More information on how the ocean was actually done on Moana:
The Algorithms Behind Moana’s Gorgeously Animated Ocean
Rigging the oceans of Disney's "Moana"
The ocean and water pipeline of Disney's Moana

Animation

The scene has minimal animation, 90 frames. In production, this shot was part of an early teaser and used to test the water. It was the opening shot, establishing the island beachfront. The camera animates slowly toward the beach, while the palm trees wave in the breeze. There is also some subtle “keep alive” animation on the foliage of certain prominent bushes and non-palm trees. The OBJ format doesn’t support animation and we didn’t feel that providing our original animation curves would be very useful, so we have baked the more prominent animation into animation subdirectories for the camera and the few elements where we felt it added to the scene. Baking the subtle “keep alive” animation on the foliage was not deemed worth the massive increase in size of the scene.

PBRT translation

We used the open source render PBRT-V3 as our data validator (the latest version as of January 29, 2018). It is free, easily accessible and the associated book is widely used. As an example of how to use our data we include the set of scripts necessary to create a complete PBRT scene. However, we have encountered some challenges which you should be aware of if you decide to use our PBRT implementation.

Art directors cannot change physics.

Early on, we had to step back from tweaking the data to look good in PBRT and just focus on the correctness of our conversion process. Much of the data that doesn’t match is because we cheat a lot. Just to name a few:

1. Internally we use two different lat-long images for environmental lighting. One for the actual light ray colors, and one for the background.
2. Our standard “quadLight” can be restricted to only illuminate certain objects. Most of the lights named beachFill or palmFill were only casting rays at those objects.
3. The position of the camera, sun and water make for a beautiful reflection. This does not match the original image with its perfectly-teal tinted water refracting through to the coral below.

Curve support.

As mentioned above, much of the data needed to recreate our many curve-based elements was lost in the process of exporting this scene. Our current curve format is translated using PBRT's new B-spline representation. However, at this time, one of the larger curve-based primitive descriptions (xgLowGrowth, associated with isMountainB) is still unrenderable due to a precision error and has been commented out in the code that generates the pbrt scene.

Our lights != PBRT lights

We have recreated the most common light type in the original shot which is basically a rectangular area light called a "quadLight". Unfortunately, the translations do not match entirely out of the box. We have corrected most of the issues related to colorspace representation (see above), but there are still subtle differences in the PBRT area-lights as compared to our original quadLights.

In addition, as noted in the "*Skydome and environment lighting"* section above, we use different environment maps for lighting. However, PBRT only allows one map, so we chose the second "visible' latlong map for use with PBRT. This means the background is an accurate match, but the lighting is a little brighter.

Clouds?

Included in the data, but not usable by PBRT is a texture map called islandsun_cloudmap.ptx. This was part of the original scene and associated with the sun_quad light where it was used as a multiplier to break up the sunlight in a "cloud like" fashion above the island.

Special handling for ptex

The PBRT geometry requires a 'faceIndices' attribute to determine which Ptex face to use for texturing. This attribute is generated by obj2pbrt when using the --ptexquads option.

Special handling for materials

In order to let obj2pbrt handle all our material bindings we ran a post process on the OBJ files to override the internal names. This left garbage "MakeNamedMaterial" definitions inside all our PBRT files with the default OBJ material. This required a post process on all the PBRT geometry files to comment those out.

Special handling for textures

At WDAS, we create all textures for an element simultaneously using an internal 3D paint program. This means each element will have a single ptex file per mesh that makes up that element, leading to hundreds or even thousands of textures for complex objects. We get around specific bindings by referring to the texture directory in that element's materials. All textures are named with the name of the mesh they belong so the directory name is all that is needed. To mimic this in PBRT, another post process script was run on the geometry file that found all shapes, looked up their materials, and then overrode their color component with a specific texture. This is why you see no textures in the PBRT material files. If a texture is missing, the object will render red.

Displacement vs bump

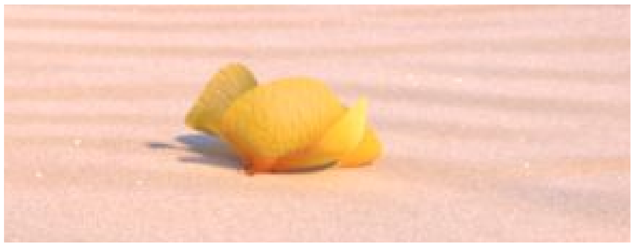
We are generating a color and an optional displacement texture for all texture-based materials. PBRT doesn't support displacements so the displacement map is being fed into the bump category. This works in some places and doesn't work in others.

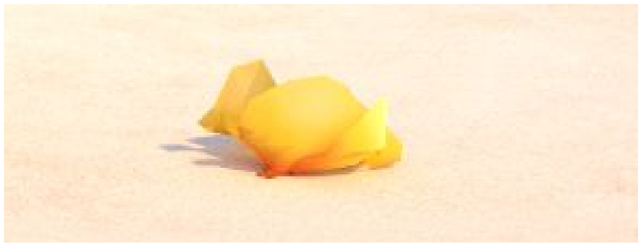
No recursive instances

PBRT does not allow for the recursive instancing. This requires us to declare all our objects up front for both the main element PBRT file and all other included PBRT files.

Subdivision

PBRT does not support Catmull-Clark subdivision and will simply render the control cages. As a result there are some artifacts in the rendered images.


[closeup of isBeach with hibiscus petals showing the limit surface (Hyperion)]


[closeup of isBeach with hibiscus petals showing the control cage (PBRT)]

Future work

There are several things which, time permitting, it would be nice to improve or add to this project. Here's a short non-exhaustive list.

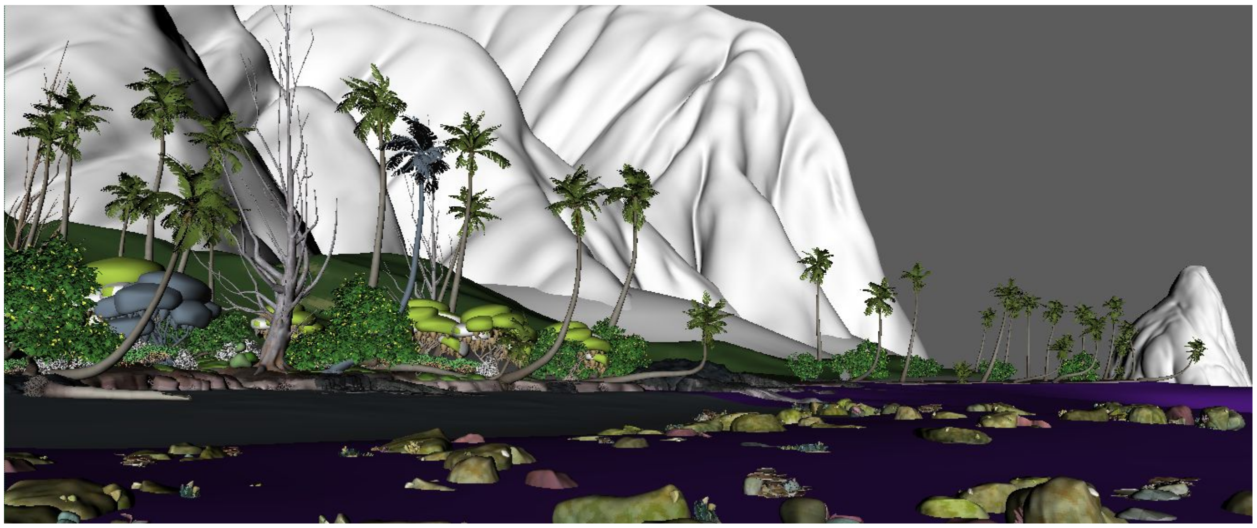
1. More complete curve data, including width and root/tip colors. Much of the visual complexity in our Disney's XGen curve-based primitives is based on how they change color from root to tip.
2. Alternate geometry and scene formats. We looked at both USD and Alembic as alternate formats, but for expediency we used OBJ. Other formats could be added in the future.
3. MaterialX files for the materials, to allow us to present our materials in more depth, with original layers.
4. Optimized textures. Most of our textures are created directly on the geometry and the user increases or decreases individual Ptex face sizes as needed. Our bake process, unfortunately, loses that optimization and produces uniformly large textures.
5. Animation support. While subtle, there is some animation in this scene that isn't currently included and which we'd like to make available. Possibly even as animation curve data.
6. Volumetric clouds. In the original shot there were painted foreground clouds. Since those clouds were composited in, they were not useful for global illumination tests. Replacing them with similar looking volumetric clouds would maintain the original artistic intent, but add something to the shot for research purposes.
7. Better curve support for bonsai elements (see ironwood). For the curve data from bonsai, we bake it out as geometry. Somehow managing to store it as curves and provides a consistent methodology to expand those would make the scene more efficient.
8. Loop the existing animation to allow longer sequences to be rendered.
9. More working code examples. Our sample code was written under tight deadlines and we were more focused on results than reusability. Cleaning it up and providing more examples could be helpful to some users.

Acknowledgments

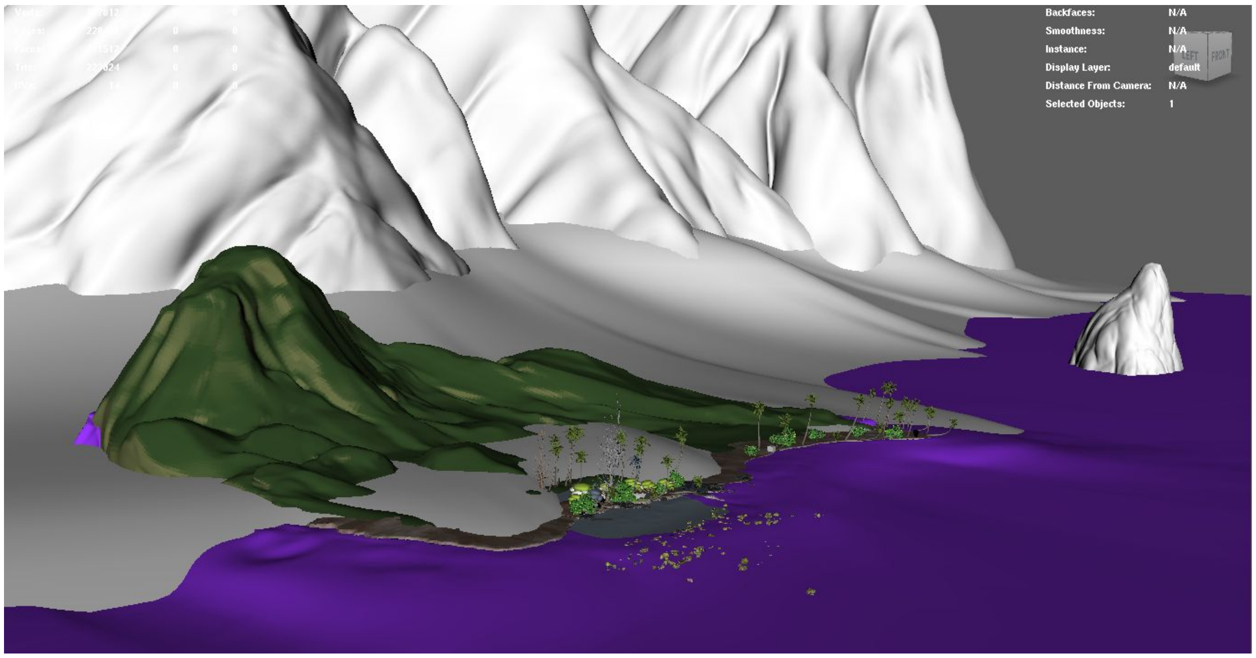
This project originated out of a request by Matt Pharr at SIGGRAPH 2016 and was championed at WDAS by Nick Cannon who got everyone to agree to let us do this. The shot was selected with input from the Hyperion team who also answered many questions along the way. Beyond that, a huge credit obviously goes to the crew on Moana for creating this shot in the first place, and also to Sean Palmer for helping resurrect the old shot as well as to Jonathan Garcia for helping recreate the ocean. Matt Pharr has been instrumental in getting the PBRT version to work, and we greatly appreciate the feedback and contribution from Intel's Embree team.

Appendix 1. Element guide

In this appendix we provide a brief overview of all the elements in the scene. The larger set pieces are the mountains, the dunes, the coachline and the beach. The images below, captured from Maya, will hopefully help you understand their relation.

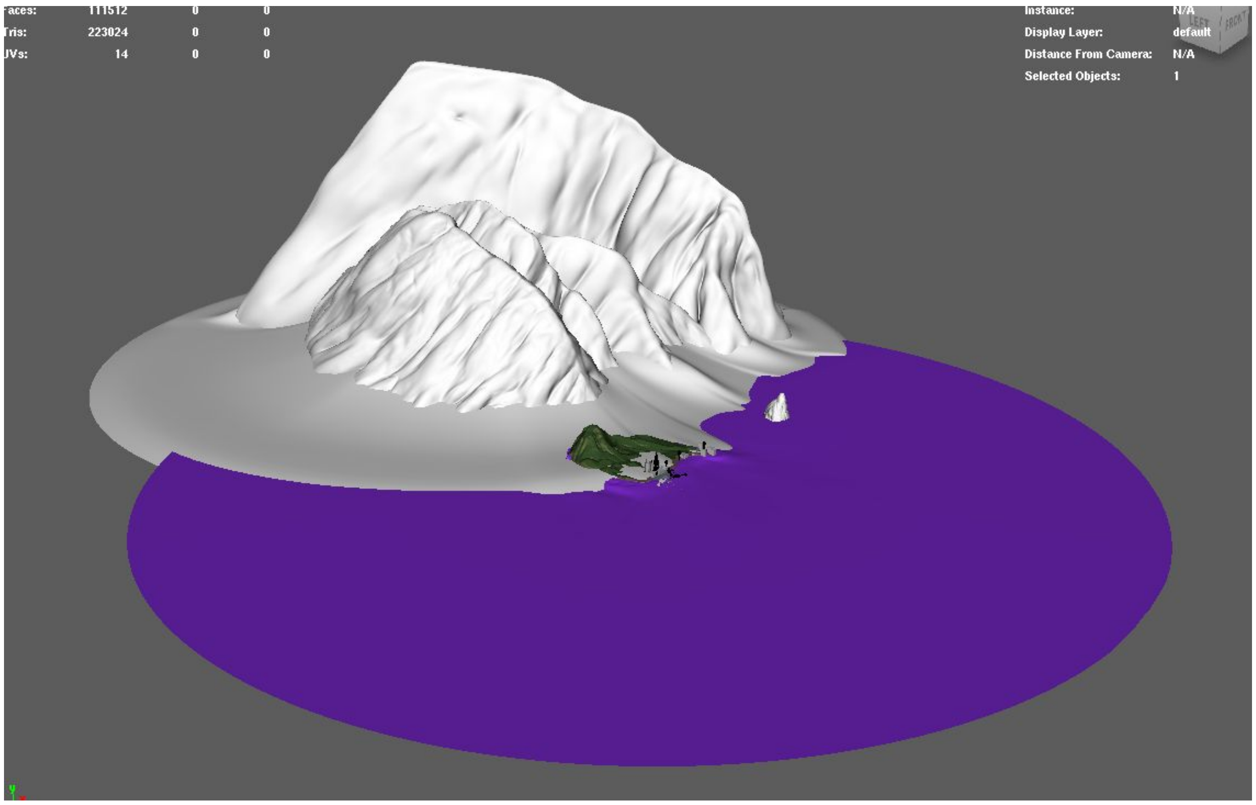


There is no Disney's XGen shown in this picture, just the base geometry and some proxy geometry. The white mountains in the back are all part of the “isMountainB” element. The solid green area in the mid-ground is “isMountainA”. The purple ground plane in the foreground is “isCoastline”. The ocean is hidden so you can clearly see the coral. The grey area in the middle is the high resolution “isBeach”, and just above it, you can see the brown visible sections of “isDunesA” and “isDunesB”. “isDunesA” is directly above the beach (left-hand side) and isDunesB is split following along the coast on either side of “isBeach”.



```
Backfaces:              N/A
Smoothness:             N/A
Instance:               N/A
Display Layer:          default
Distance From Camera:   N/A
Selected Objects:       1
```

Pulling back we can see just how large isMountainB is in relation to isMountainA. You can also see the “isCoastline” continuing back and becoming the ocean floor.



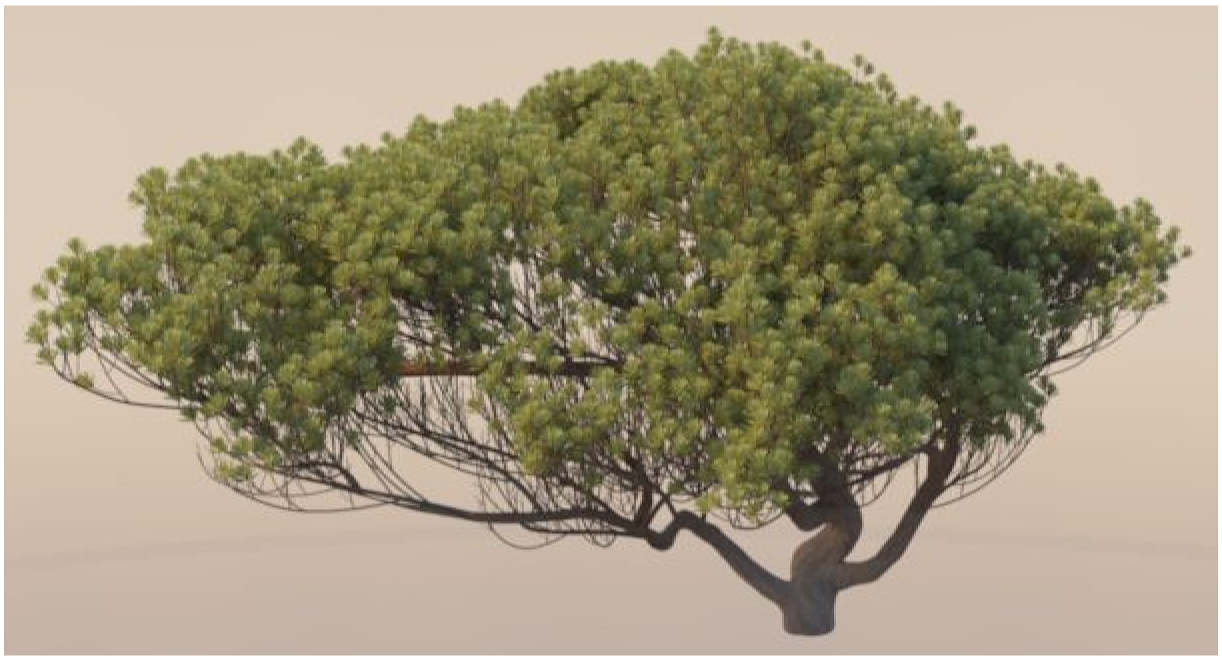
```
aces:    111512    0    0
Tris:    223024    0    0
UVs:         14    0    0
```

```
Instance:               N/A
Display Layer:          default
Distance From Camera:   N/A
Selected Objects:       1
```

The world view.

isBayCedar

Originally a bonsai element, we converted the leaves to the instance format and baked all the branches into the main geometry. The trunk and leaves have texture maps and there is displacement on the trunk.



[isBayCedarA1 rendered in Hyperion.]

isBeach

This is the high resolution beach that was used for several closeup shots in the test sequence. It contains numerous primitive descriptions used to create beach debris as well as one of the largest textures to convey the sandy appearance for close up shots.

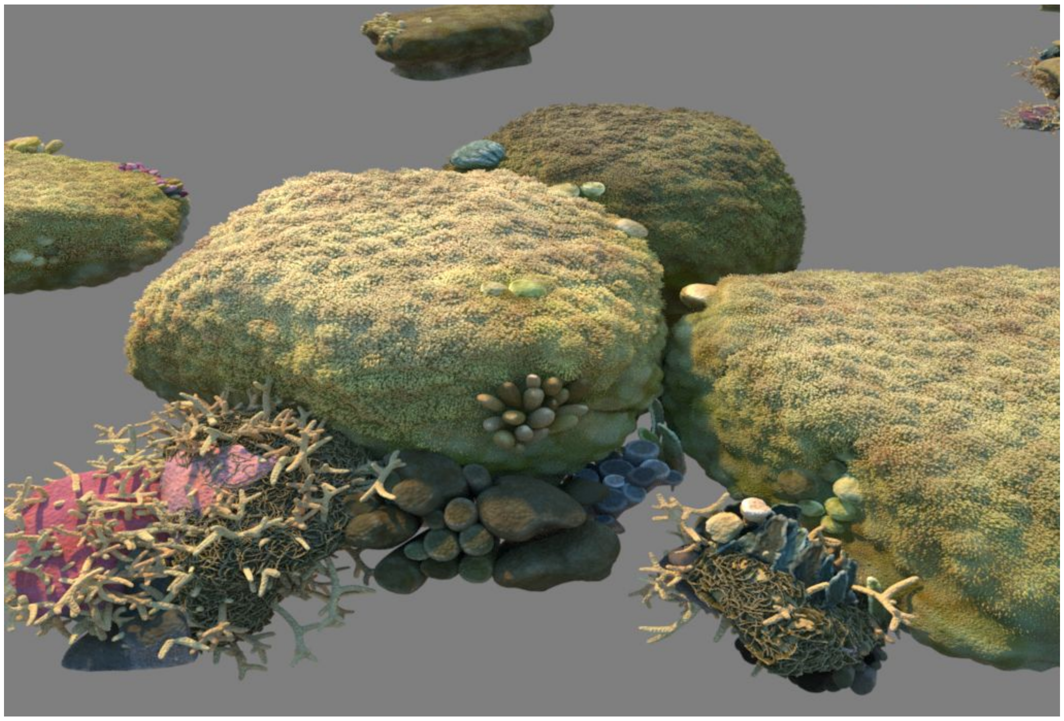
isCoastline

The coastline geometry is mostly hidden in the shot, lying either under the ocean or contributing small bits of grass and root along the distant shore.

isCoral

The coral element is an example of copies with distinct geometry and primitive files. To improve the look of the lagoon, layout artists took each individual coral and moved clusters around to create a more random distribution of coral. Since we don't have sub-element transforms in our simplified baked versions, each copy of the element has its own individual geometry file with those internal transformations baked in.

In addition, the coral element, specifically the parts of the model covered with the "xgFlutes" primitive description, have lost visual variety in the translation. The single archive associated with "xgFlutes" is a small cylinder with a dimple in top, which looks a little like a golf-tee. The three large rocks below are completely covered with this archive, which is scaled to a near sub-pixel size and distributed in a manner that produces a larger scale bumpy pattern. Each archive then pulls its color from a mottled map based on the original rock surface. While we are able to maintain the position of the archives, we lose the color detail and instead have a single color per rock.

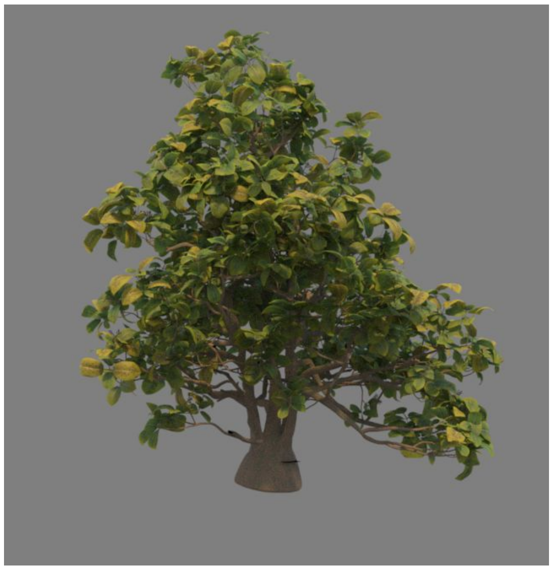

[detail of isCoral rendered in Hyperion]

isDunesA & isDunesB

The dunes are the transition element between the beach/coastline and the mountains. The isDunesA is the more detailed surface as it sits directly above isBeach. Both elements contained modeled dunes of soil that represented the eroded portion of the island that is not touched by the ocean. In addition, both dunes have various primitive descriptions that contain larger foliage and trees designed to blend in and hide the lower resolution mountains.

isGardeniaA

This is another bonsai element in which the procedural branches were baked into the main element while the trunk and the instanced leaves are defined by a primitive description.

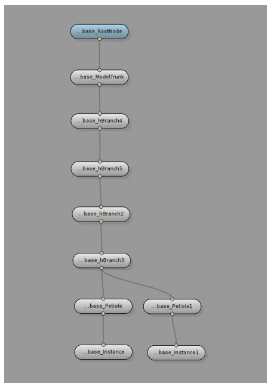


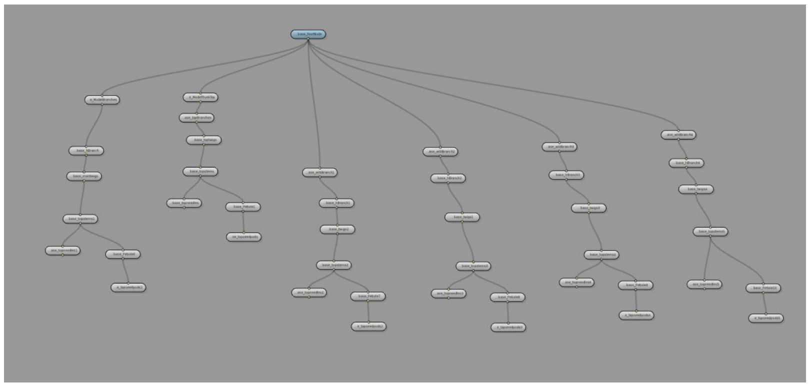
isHibiscus and isHibiscusYoung

This another very standard bonsai element, consisting of an model trunk, procedural branches and instanced leaves and flowers. The hibiscus flowers are also reused on the isBeach element and we made some effort to share textures.

isIronwoodA1 and isIronwoodB

The ironwood trees proved to be one of the heaviest single elements and one of the of the most complex. It is a bonsai-based structure, like most of the trees. But, unlike other trees, it is comprised almost entirely of curves and has a considerably larger branching structure.




[isHibiscus branching structure in bonsai vs isIronwoodA]

Our established paradigm for dealing with bonsai had been to instance the leaves and bake down the branch structure, which worked well for the rest of the trees, producing large, but not unreasonably so, branch files. This was not the case for the ironwood trees. Both primary ironwood obj files are around 3 GB each, for just the branch structure. This large size made them particular unwieldy when it came to both creation and translation to other formats.

[Hyperion render of isIronwoodB]

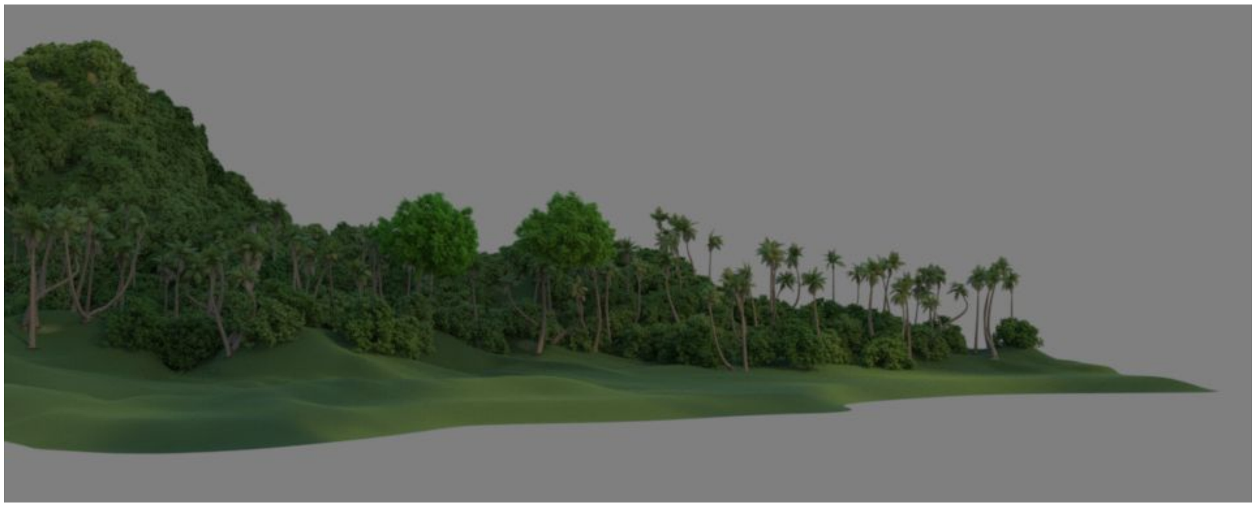
isKava

Another basic bonsai shrub with a modeled trunk structure, baked branches from procedural curves and instances leaves. This is one of the smaller elements and is used mostly in the hero beach area to fill in gaps.

[Hyperion render of isKara]

isMountainA

The isMountainA groundplane represent the midground foliage that blends between the foreground isCoastline and the background isMountainB. It is a relative simple model that is covered with large pushes to hide empty areas and background palms tree to add to the silhouette.

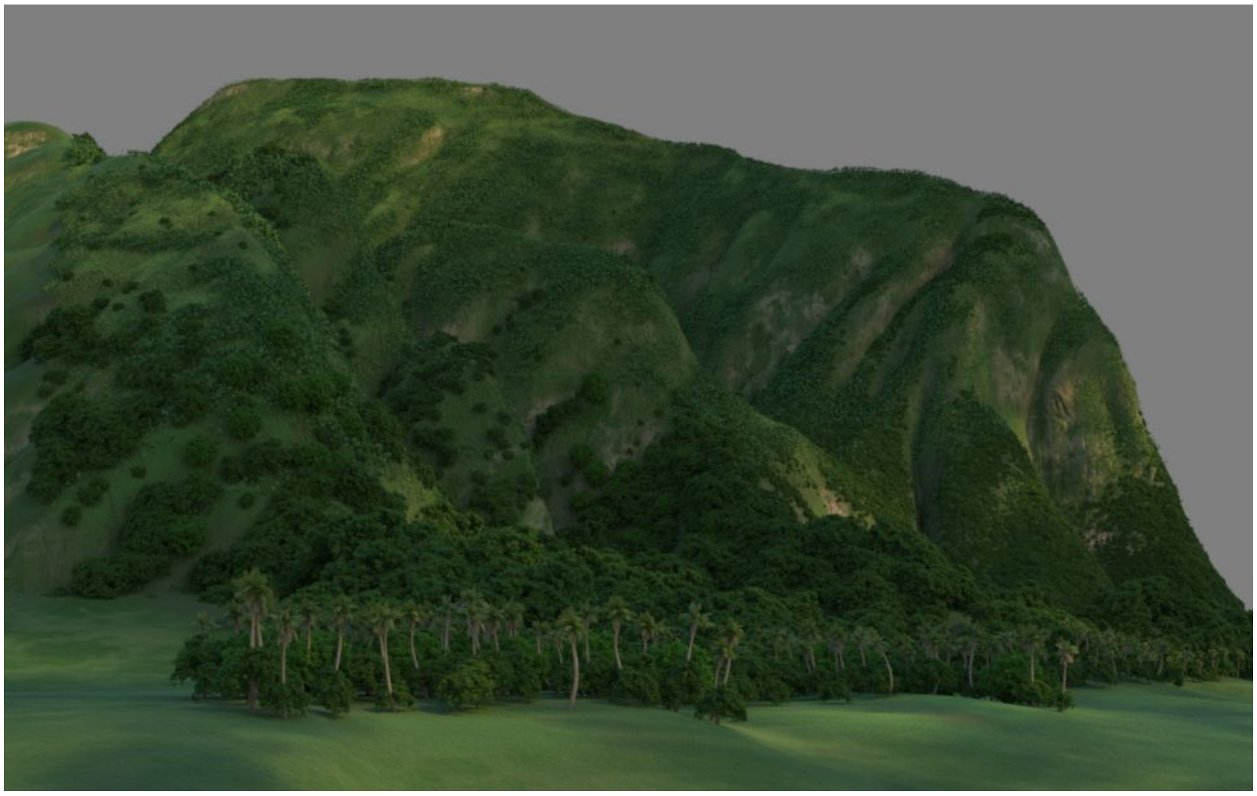


[Hyperion render of usMountainA]

isMountainB

The isMountainB groundplane represents the farthest land in the shot. The majority of its foliage is groundcover-live clusters of leaves that are designed to lend a sense of depth and distance to mountains hovering above the beach. At the foot of these mountains are more

The xgLowGrowth primitive description is just a single curve instanced 5 million times, making it the largest curved-based primitive description and the second largest primitive description by instance count.



[Hyperion render of isMountainB]

isNaupakaA

The is a bonsai element designed as a ground cover. It presented some challenges in that it was designed without a trunk. All the instanced leaf components are growing off a collection of flat puddle-like surfaces that were basically a proxy stand-in for were the Naupaka would be in the final scene. Since our workflow separated the main geometry file from the instanced primitives, we found ourselves with what was essentially an empty geometry file after we removed the proxy. We didn’t want to break the system, so we left the proxy in the file and assigned it a “hidden” material to flag users to not render it.

[Hyperion render of isNaupakaA]

isPalmRig

One of our more complex modelled trees, the palms have a trunk, root structure, and considerable modelled detail about where the “frond spines” come out. These spines are the central core for each palm leaf, which is curve instanced geometry, each leaf being made up of dozens of flat curves, including the dead ones hanging down.

[Hyperion render of isPalmRigA3]

isPandanusA

The pandanus trees are Disney's XGen-based, not bonsai. Similar to bonsai trees, they have a modelled trunk, but their trunk is more extensive since they do not have the procedurally generated branch structure that bonsai creates. The XGen is driving the leaf creation, which are all thick clusters of curves, representing the long, flat leaf that the plant is famous for. Another oddity of the pandanus is its adventitious roots, which, at first glance, might seem like bad geometry.

[Hyperion render of isPandanusA and copies]

Appendix 2. Data formats in-depth

Element Json File

The primary scene file format is json. We chose it for its readability and the ease with which it can be accessed in python. Each element has a single json file that describes the primary element data. The file is stored in island/json/{element}/{element}.json.

The primary names/keys of the element json file are:

name: A string with the name of the element, e.g. “*isBayCedarA1*”

geomObjFile: The OBJ file name that represents the element, excluding any instanced primitives, e.g. "obj/isBayCedarA1/isBayCedarA1.obj". For most plant elements, this is the trunk, as the leaves are instanced. For ground plane-like structures, this is the geometry that makes up the ground itself.

matFile: A json file name containing the materials associated with this element, e.g. “json/isBayCedarA1/materials.json".

transformMatrix: A float[16] array representing the transform matrix to be applied to the element. In cases where the element is copied, this is applied to the first occurance of the element (the copies will have their own transforms). For most of the ground plane-like structures, this is the identity matrix.

instancedPrimitiveJsonFiles: This top-level structure represents the various primitives associated with an element. In terms of Disney's XGen, these are stand-ins for XGen description, with each key into the structure representing a single description. For a simple tree or plant, this might be ‘xgLeaves’. For bonsai-based element, this is names ‘xgBonsai’. For a ground plane, there would be numerous primitives that could represent all the plant life and debris that covers the plane.

For each of these descriptions, the secondary names/keys are:

jsonFile: A json file name that contains the data needed to create this archive. The exact contents of this file vary depending on the type.

type: A string indicating one of the three types of primitive descriptions we support:
curve: This description is a series of curves. For example, grass or roots.
archive: This description is a series of instanced geometry files. For example, sea shells, beach debris, or plants.

element: This description is another element that is being referenced. Many of our large, distant ground planes are covered with trees, identical or nearly identical to other tree elements we've already defined. To prevent redundancy, we are referencing that other element's data on disk, allowing us to reuse geometry, textures and materials.

archives: Only present in primitives of type archive or element. A list of all geometry files used by this description.

element: Only present in primitives of type element. The referenced element associated with these instanced primitives. That element's materials and textures should be used instead of the primary element's.

variants: Only present in primitives of type element. A variant is a slightly different geometry file that uses the same materials and textures. Use this variant name for the associated primitive element.

widthTip: Only present in primitives of type curve. For curves, this is the width of the tip.

widthRoot: Only present in primitives of type curve. For curves, this is the width of the root.

degrees: Only present in primitives of type curve. The degrees in the curve.

faceCamara: Only present in primitives of type curve. The curve should be considered to always face camera for a consistent width. When a curve is a stand-in for a flat surface, like grass, this should be false. For a rounded surface, like hair or roots, this would be true.

Here is an example of a single instanced primitive of type element defined within an **instancedPrimitiveJsonFiles** structure associated with *isDunesB*:

```
"xgTreeFill": {
     "jsonFile": "json/isDunesB/isDunesB_xgTreeFill.json",
     "variants": [
         "base",
         "bonsaiA",
         "bonsaiB",
         "bonsaiC"
     ],
     "type": "element",
     "archives": [
         "obj/isBayCedarA1/isBayCedarA1.obj",
         "obj/isBayCedarA1/isBayCedarA1_bonsaiA.obj",
         "obj/isBayCedarA1/isBayCedarA1_bonsaiB.obj",
         "obj/isBayCedarA1/isBayCedarA1_bonsaiC.obj",
         "obj/isBayCedarA1/archives/archivebaycedar0001_mod.obj"
     ],
     "element": "isBayCedarA1"
 },
```

instancedCopies: This top-level structure represents copies of the current element placed around the scene. An excellent example is *isBayCedarA1*, which has 8 additional copies at various locations in the scene. The name/key into the structure is the 'name' of the copy, which is usually the original element name with a number at the end, i.e. *isBayCedarA2*. The secondary names/keys are:

> **name**: Redundant occurance of name.
>
> **transformation**: This transformation matrix replaces the top-level transformation matrix.
>
> **geomObjFile**: This copy is using a different geometry file, but still the same textures and materials. The *isPalmRig* elements are good examples of 20+ copies of the same element, each posed into a different position, e.g. " obj/isPalmRig/isPalmRig14.obj"
>
> **instancedPrimitiveJsonFiles**: Overrides the top-level instanced primitive description with this one. An example of an instanced primitive structure within an instancedCopies structure that overrides the top-level values:

```
"isPalmRig14": {
    "transformMatrix": [
        -0.058, 0.0, -0.65, 0.0,
        0.0, 0.65, 0.0, 0.0,
        0.65, 0.0, -0.05, 0.0,
        5662.6, 34.15, 635.63, 1.0
    ],
    "geomObjFile": "obj/isPalmRig/isPalmRig14.obj",
    "name": "isPalmRig14",
    "instancedPrimitiveJsonFiles": {
        "xgFrondsA": {
            "type": "curve",
            "widthTip": 0.4,
            "jsonFile": "json/isPalmRig/isPalmRig14_xgFrondsA.json",
            "widthRoot": 1.75,
            "degrees": 2,
            "faceCamera": true
        }
    }
},
```

variants: This is the first top-level structure found in element json files. In many ways it is very similar to the **instancedCopies** structure in that is is a series of element overrides attached to a variant name. The primary difference is that **instancedCopies** are referenced by the element file, referring to copies of itself whereas **variants** are references by other element files as an instanced primitive of type element. The only elements that have variants are *isBayCedarA1* and *isIronwoodA1*. These are both referenced in *isDunesB*. All secondary name/keys that are valid for **instancedCopies** are valid for **variants**. If we were to refactor the json format, we'd combine these two structures.

Here is an example of a variant defined for *isBayCedarA1* and references in the json example for *isDunesB* at the top of this section.

```
        "bonsaiA": {
             "geomObjFile": "obj/isBayCedarA1/isBayCedarA1_bonsaiA.obj",
             "instancedPrimitiveJsonFiles": {
                 "xgBonsai": {
                     "jsonFile":
"json/isBayCedarA1/isBayCedarA1_bonsaiA_xgBonsai.json",
                     "archives": [
                         "obj/isBayCedarA1/archives/archivebaycedar0001_mod.obj"
                     ],
                     "type": "archive"
                 }
             }
```

Element primitive json file

These files represent the primitive data that is created for each element. They are named json/{element}/{element name}_{primitive}. Element name can be either the element, the name associated with a copy, or the name associated with a variant. Primitive names start with 'xg'. The contents of the file will vary depending on the primitive type :

curve: A json list of curve data. Each curve is composed of float[3] arrays to represent the uniform b-spline data for 2nd degree curves.

archive A json dict. The top level key is the specific archive filename. The secondary key is distinct name associated with that instance of the archive, and the final value is the transform applied to that particular instance. Here is an example from *isBeach*, isBeach/isBeach_xgHibiscus.json, which has 8 total top level keys, representing the 8 different Hibiscus models that are scattered around the beach.

```
          "obj/isBeach/archives/xgHibiscus_archiveHibiscusFlower0006_mod.obj": {
               "xgPrim176_xgHibiscus": [
                   -0.56, 0.09, -0.94, 0.0,
                   0.004, 1.099, 0.11, 0.0,
                   0.95, 0.05, -0.56, 0.0,
                   229.4, 1.45, -11.04, 1.0
               ],
               ...
```

element: A json dict. This is format similar to the archive type. The top level key is the name of the variant of the particular element that is being referenced. The obj file for this element is defined within that element's main json file. The secondary key is a distinct name associated with that instance of the element and the final value is a transform. Here is an example from *isDunesB*, isDunesB/isDunesB_xgTreeFill.json. The file contains 3 variants and the primary element of isBayCedarA1. The keyword 'base' is used to signify that the variant is the main one.

```
"bonsaiA": {
     "xgPrim106_xgTreeFill": [
         -0.02, -0.008, 0.82, 0.0,
         0.03, 0.82, 0.01, 0.0,
         -0.82, 0.03, -0.02, 0.0,
         -801.9, 28.21, -480.0, 1.0
     ],
     ...
```

Material json file

There is one material json file per element, which is handily named material.json, although it's also defined in the element json file. The top level key is the material name. This file is generated directly from the artist's file used in production and lacks a uniform material naming scheme. The secondary key/value pair is the Principled Material parameter and its value. We used the names from the latest paper to minimize confusion. In addition, the following parameters have things to note about them:

baseColor: This is the color associated with the material. In places where this should be replaced by a texture, this color is usually (but not always) red.

colorMap: This string is a directory that contains one Ptex file per mesh that the surface is bound to. This single layer has been baked down from the multiple layers that are contained in the artists data.

displacementMap: This string is a directory that contains one Ptex file per mesh that represents the surface displacement. This single layer has been baked down from the multiple layers that are contained in the artist's data and includes all multipliers baked down as well.

assignment: A json list of surfaces names that this material is bound to, generated from the original artist data. This should be redundant data as the material for each mesh is embedded within the OBJ file. Should you use this data to generate your own bindings, please be aware that you need to apply the original hierarchy to the mesh (stored in the .hier file) as well as check again the primitive description name. See the Caveat "OBJ Format and Material Binding" above for more details.

type: "thin" or "solid" to represent a BSDF vs a BRDF, respectively.

Lights json file

The lights json file contains a single dictionary with one entry per light, keyed on the name of the light in the original production shot.

Each light entry has the following data associated with it:

translationMatrix: float[16] representing the translation matrix of the center of the light. These are based off Maya worldspace coordinates.

rotation: float[3] representing the maya rotation transform as a separate value.

location: float[3] representing the maya rotation transform as a separate value.

color: The color of the light source.

exposure: An exponential multiplier for final light brightness. radiance = 2^exposure * color^2.2

type: The two types of lights in the scene are either 'quad' or 'dome'. The quad light is a large rectangle that radiates energy. It is used for all of the direct lighting in the scene. The second type of light is a 'dome', which provides texture maps that can be mapped to infinity rays. These rays can either provide an end color (acting like a 'background') or can illuminate as well.

height: The height of the radiating rectangle associated with a quadLight.

width: The width of the radiating rectangle associated with a quadLight.

map: The latlong used for image based lighting. Only present in dome lights.

envmapCamera: A latlong used for the background.

The primary lights in the scene are "sun_quad_llc" and "sky_dome_llc". All other lights are added for art direction purposes or to provides a physical reaction that wasn't present in our initial test scene. Specific examples would be bounce lights on several palms to fake light reflected off the addition or key lights to brighten up certain areas so they stand out from the rest of the environment.

Camera json file

The camera file format is a single camera definition per json file. Much of this same is written directly out of maya based on the maya 'persp' camera representation. The keys are:

name: The name of the camera. Should match images stored in the ref/ directory.

eye: World coordinate position of the camera (MFnCamera.eyePoint).

up: Camera ‘up’ vector (MFnCamera.upDirection).

look: Traditional CG ‘lookAt’ vector (MFnCamera.centerOfInterestPoint).

fov: Field of view of the camera. (MFnCamera.horizontalFieldOfView).

ratio: Ratio of the final movie rendered frame, width/height, 2.38.

focalLength: Focal length of the maya perspective camera obtained via mel.

centerOfInterest: Center of interest of the maya perspective camera obtained via mel.

lensRadius: Calculated from OpenMaya values for focal length and f-stop.

screenwindow: Calculated from OpenMaya screen shift values and ratio.

Hierarchy file

The hierarchy file is used to store the original maya hierarchy which is discarded during the OBJ conversion. Our hierarchy structure is fairly important to our binding scheme, which often binds at group nodes instead of the final mesh name. The file is a very simplistic json file containing a list of meshnames and their group hierarchy. The file is stored alongside the obj file it is associated with and has the name basename. Here is “isLavaRocks.hier”:

```
{
   "rockfacemain0001_geo": "|isLavaRocks|trs_master|trs_shot|trs_aux|model|main_part|herorocks0001_mod",
   "rockfacebg0001_geo": "|isLavaRocks|trs_master|trs_shot|trs_aux|model|main_part|herorocks0001_mod"
}
```

The basic format for each line is “mesh name” : “|group|group|....|group”. You will notice immediately that most of our elements have a standard common hierarchy that incorporates the element name at the head:

|*elementName*|trs_master|trs_shot|trs_aux|model

Under the model node is when the differentiating group structure begins for each element.