Scene Semantics Applied to Dynamic Frame Generation

Mark Wesley Harris
CS5800 Computer Graphics, Fall 2019
University of Colorado Colorado Springs
Colorado Springs, United States
wharris2@uccs.edu

***Abstract*—We discuss our research on the relationship between a 3D scene and the rendered frame it produces, denoted as the "semantics" of a scene. Topics of interest include path tracing, shaders, and color theory as applicable to RenderMan and Autodesk Maya. We modeled and animated a dynamic scene, and implemented Python scripts to encode and decode semantic data stored in RenderMan shaders. Our results showed how certain data could be stored at rendertime, and processed later as necessary. This research is applicable to computer visualization and other deep learning problems in Computer Graphics.**

I. Introduction

Identifying the relationship between a complex 3D scene and its rendered frame is very difficult for a computer to accomplish. Even though the scene exists in the digital world, this kind of abstract data is not easily extractable. We use the term "scene semantics" to define this problem – scene semantics are a description of how a complex 3D environment affects what is ultimately rendered. This study is key to understanding and perhaps improving the render pipeline and current rendering techologies.

We successfully implemented a system capable of encoding and decoding semantics for a 3D environment in an understandable way. Here we ellaborate on our research and the results of our implementation. First we describe our preliminary research on path tracing and the render pipeline. We then detail the setup of the test environment, implementation, and our results for different precisions. Finally, we discuss the generated semantics and how they can be used to solve complex problems in Computer Graphics.

II. Research

Our first task was to examine what information could be generated from the 3D scene. There is ample information available, however we must select only the most relavent for creating usable semantics. Our first approach to this problem was in researching some of the most basic (and powerful) components of the render pipeline, path tracing and shaders. The fundamentals discussed here are important to understand our dicisions for implementation, discussed later in Section IV.

A. Ray Tracing and Ray Marching

Ray tracing is the study of how light behaves in a given environment. Light behavior can range from simple intensity calculations to complex reflections and refrations. Figure 1 shows one of the first studies of ray tracing, where light rays were mapped from the viewer to a light source. Avro *et al.* discuss the difficulty of this problem, as it involves taking into consideration material properties, light sources, and where the viewer is looking [1].

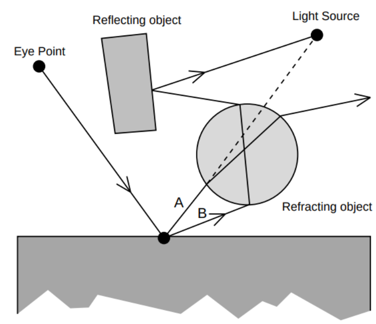


Figure 1: Example of a problem in ray tracing [1].

Technological advancements since 1986 greatly improved ray tracing capabilities, and the improvements are well documented. For instance, the ray tracing walthrough created by Scratchapixel provides simple code samples for casting rays from the camera, and other basic ray tracing functionality [2]. From these advancements also came the concept of ray marching. Ray marching is a derivative of ray tracing, and is used to model interactions of light through volumetric surfaces. Figure 2 shows an example of how ray marching is applied to create volumetric shaders, such as smoke or fog [3]. Examples of both ray tracing and ray marching effects can be seen within the rendered scenes shown in Figure 3 and Figure 5b.

B. RenderMan

Two renderers that are now highly developed are the Arnold Renderer [4] and the RenderMan renderer [5]. Each renderer functions differently, but in general provides different interfaces for the same tasks. RenderMan 22 – which is maintained by Pixar Animation Studios – was chosen to be the renderer software for the purposes of this project, since it is arguably the most advanced renderer developed to date. Figure 3 shows two examples of dynamic scenes rendered with RenderMan, one from Jurrassic Park and the other from Terminator 2 [6].

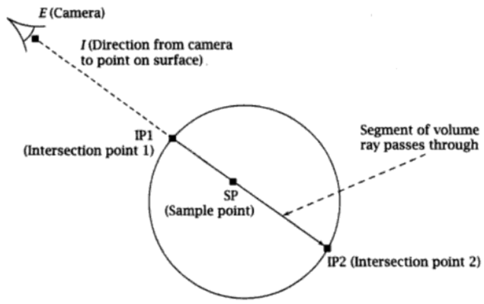


Figure 2: Ray marching technique for volumetric shading [3].

RenderMan includes many resources via their documentation, however outside of officially produced documents there is an absence of tutorials or walkthroughs for shaders in recent RenderMan systems. Introduction materials provided by Pixar were used to research the capabilities of RenderMan shaders and the rendering pipeline [7]. We found that the RenderMan Interface provides implementations for rendering "…hidden surfaces, spatial filtering, dithering, motion blur, depth of field, flat and curved surfaces, objects, constructive solid geometry, and programmable shading to express lighting conditions, shadows, and surface appearances, with sophisticated control over color, texture, and reflectivity" [5]. Many of these attributes were out of scope for this project, however they may later be explored in order to test the capabilities of what was developed. We were interested in how to harness RenderMan's shaders with the power of path tracing to produce scene semantics with the desired levels of relevance and detail.

Figure 3: Path-traced images rendered with RenderMan [6].

RenderMan shaders are written in one of 3 ways: Patterns, Open Shading Language (OSL), and C++ [7]. Patterns are useful for adding noise to materials, or generating source data programatically. Examples of source data include fractals, shapes, gradients, and mult-layered noise, which can be combined in different ways to create unique images. Patterns can be used to create rust, wavy glass, or vector-based shading. Open Shading Language (OSL) is a programming language artists use to create more refined shading effects. "RenderMan Shading Language (RSL) …includes math operations (sin, sqrt, etc.), vector and matrix operations, coordinate transformations, and higher level functions like noise and texture" [5]. Shaders for RenderMan are written in RSL (derived from OSL) or C++, and are attached to different objects via materials. C++ implementations are more complex, but have been proven to run faster in some situations [7].

In any of the three cases, shader outputs are not written directly from shaders to the rendered frame. They are instead handled through a BxDF material (the most basic BxDF material for RenderMan is called "PxrSurface"). When a scene is being rendered, data from all visible materials is combined and sent to a unit called the Integrator, which converts the data into pixels for the rendered frame. Integrators can be hot-swapped for different types of rendering and debugging of shader code.

Upon researching how ray tracing is handled, we found that information on refracted and reflected rays is passed back to the BxDF for recursive sampling, and processing continues until the render has sampled the rays in sufficient detail. Anti-aliasing and other filters can be added at the end as necessary to produce the desired frame. Figure 4 shows an abstract summary of the RenderMan render pipeline.

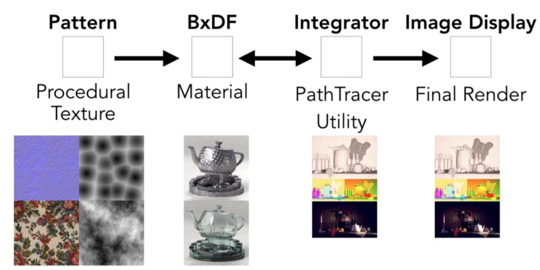


Figure 4: Data interactions between shaders and the rendered frame [7].

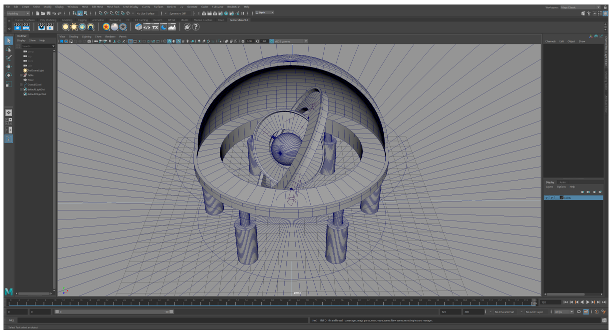
Shaders have complex interactions with data during ray tracing, however it is unclear how to extract the data for later use. A custom integrator could be written to collect the BxDF data, but even then it could lose the connections with the scene that are required for semantics generation. While path tracing has powerful capabilities, we found it is infeasible to extract the raw data during processing at the shader level. Although obtaining data from within shaders is impractical, a shader's inputs and outputs can be used to store semantic data. This approach disregards the benefits of ray tracing or ray marching for the time being, but retains the concept of encoding data into shaders that can later be used for generating semantics.

III. Environment Setup

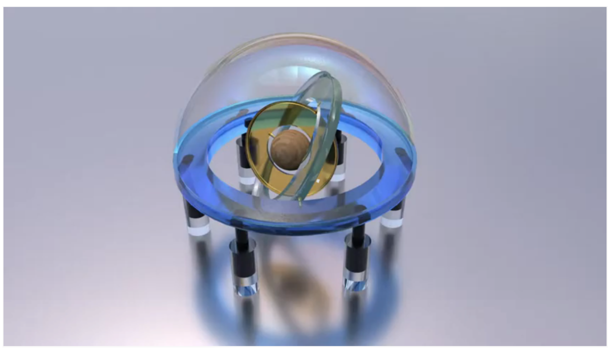
Scene semantics requires we look at complexities ranging from simple to dynamic – a powerful computer and complex source animation were required to produce meaningful research on this topic. Described here are the steps taken for setting up the project environment and preparing for our research, development, and evaluation.

A. Modeling

The modeling and animation software utilized was Autodesk Maya 2019. The model created for this project is shown in Figure 5a. To model one ring of the table, a primitive cylinder was flattened. Its rim was extruded, and the inside raised slightly to create a more interesting object. This new disk primitive was copied twice in order to form the two inner-portions of the table. Sphere primitives were used to create the half-dome on top of the outer-table, and the centerpiece that rests inside. 12 cylinders were used for the legs, and cylinders were also used for the hinges that join the tables together.



(a) “Table” source in Autodesk Maya.



(b) “Table” rendered with RenderMan [8].

Figure 5: Autodesk Maya and RenderMan environments.

Each object was assigned a material. As discussed in [7], RenderMan comes with a shelf of unique material presets that may be further customized as desired. Dynamic rendering was prioritized in the selection of materials – instead of choosing one material to customize for each part of the table, many different materials were used in order to obtain the most dynamic render possible. All components besides the floor and center sphere were transparent, with varying indices of refraction. The result was a complex scene with a considerably long render-time. Rendering a single frame of resolution 1920 x 1080 took around 15 minutes to complete.

B. Animation

In order to animate the scene, joints were placed at each hinge and parented to the object groups they operate. Control objects were then created and constrained for each joint group, since it is vital that the joints themselves were left unaltered throughout animation [9]. In total, 5 control objects were used to animate the joints in the scene. Keyframes were added over the course of 120 frames, and made to loop smoothly.

Since rendering a single frame took approximately 15 minutes, rendering the entire 120 frame sequence required around 30 hours total. The resulting sequence of frames created an animation around 5 seconds long, but with plenty of dynamic material interaction to be used in later research. A sample frame of the final animated sequence is shown in Figure 5b. The animation itself is posted on YouTube [8].

IV. Implementation

We decided to implement a system where semantics for each frameblock can be generated at rendertime, but stored for later post-processing. Data is stored visually to keep the render pipeline free from unnecessary frameblock calculations. The process first involves screen segmentation in order to assign color masks to each frameblock. We then wrote scripts to encode and decode objects using this concept, and finally generated semantics per object in each frameblock.

A. Screen Segmentation

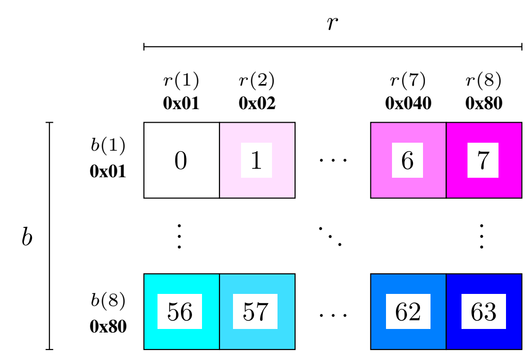


Figure 6: Parameterization of screen space using r and b.

We expand on the concept of frameblocks explored in [10] to encode scene semantics in a visual way. Screen segmentation can be used to assign a unique color to each object in the scene based upon which frameblocks it resides in. In order to show this information visually, we exploited the red (r), green (g), and blue (b) color channels of a rendered image. A primitive segmentation of screen space is shown in Figure 6, where r and b were used to parameterize the entire screen and create a map of color masks to frameblocks. Each frameblock is given a unique binary color mask based on its position in the segmented screen. Assuming integer representation of colors, values can use up to 8 bits of binary encoding. We let each of the 8 x 8 blocks parameterized by r and b represent one bit mask for either color. Each frameblock then has a unique (r, b) color-code with possible component values 0x01, 0x02, $\ldots$, 0x04, 0x08. We can use a logical disjunction to “add” blocks together, for when an object resides in more than one frameblock.

A resolution problem arose from this method, since the screen can only be broken into 64 blocks total. Our frames have a screen resolution of 1080 x 1920, meaning each block has a resolution of 135 x 240. The desired resolution is much smaller, 64 x 64 or even 32 x 32. We decided to take advantage of the third color channel, g, to help solve this dilemma. First we considered using each of the 8 bits of g to overlap with one parameterized section of screen space. The screen is then broken into 4 x 2 sections of 64 frameblocks, or 512 frameblocks total. Now each frameblock has a resolution of 67 x 60.

This has the desired resolution, however we find there is yet another problem – if an object spans multiple 8 x 8 frameblock sections (the odds of which increases the smaller the frameblocks become), then it is ambiguous which frameblocks are active and which are not within a frameblock section. This new dilemma can be solved by offsetting the domains of g so that, instead of covering frameblocks $r_i(0..8), b_i(0..8)$, it covers $r_i(0..4), b_i(0..4)$. Each 4 x 4 frameblock section has some wiggle-room built in around it, and therefore has more precision. The final partition of the screen is shown in Figure 7. Notice that r and b are subscripted, to show where each frameblock section begins and ends.

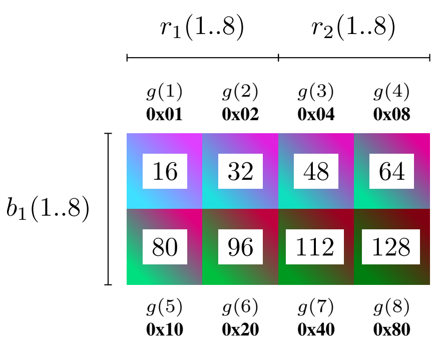
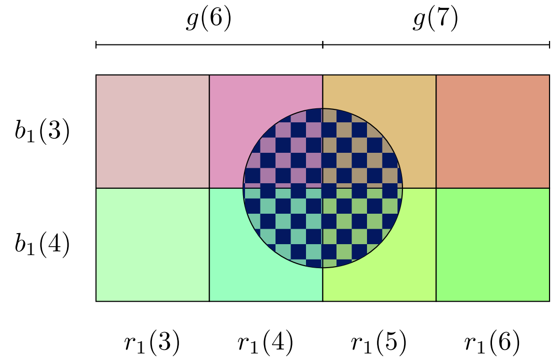
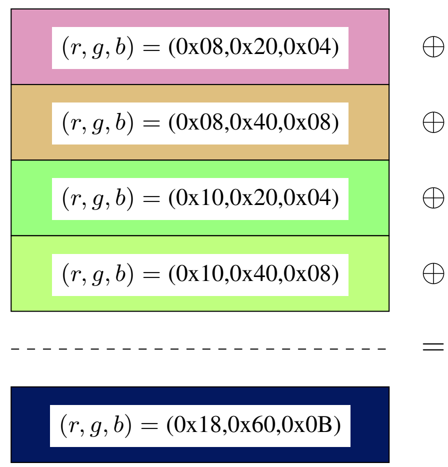


Figure 7: Final partition of screen space, where g is overlaid on top of the r and b parameterization.



(a) Example of an object overlapping a partition.



(b) Derivation of the color for the example object.

Figure 8: Example showing how a color is selected for an object straddling more than one frameblock. The color of the circle is made up of the logical disjunction of color bits.

This solution divides the domains of g in half, meaning we can only cover 128 frameblocks instead of 512, and blocks now have resolutions of 135 x 120. An example calculation of the final segmentation system is shown in Figure 8. This proves that, although there are less frameblocks than the 512 case, the accuracy of the colors increases for objects spanning frameblock sections.

B. Optimizations and Other Considerations

While the 8-bit model is useful for integer storage of color values, higher precision models can be applied to shrink frameblock resolution even further. Using floating-point values, we can have up to 64 bits of precision. While this is great in theory, the increased precision makes it harder to distinguish objects from each other at rendertime, and requires many more bits to store the encodings.

We decided to test 8-bit and 16-bit precisions for our example scene. 16-bit precision grants us frameblocks of sizes 33 x 60, which fit our desired frameblock resolution (albeit frameblocks are now more rectangular). Renders for both precisions are shown in Figure 10. Comparing the two, we see that 8-bit precision offers a clearer representation of the color spectrum over the 16-bit model. This is by the nature of binary encoding, since color values are halved at each successive bit. We can, however, use the 16-bit color values given precise enough color detection or if the raw shader outputs are available. If neither option exists, the only possibility is to use the 8-bit precision, with 135 x 120 resolution frameblocks.

C. Final Semantics Generation

We must now decide what data to collect and export for each frameblock. If we collect enough information about each object, we should be able to discern with some amount of clarity the relationship between the object and rendered frame. The following information for each object in a frameblock was considered:

d – Distance to center of frameblock.
t – Translation of object.
r – Rotation of object.
s – Scaling of object.

Line Number	Description
184 - 207	Split the screen into a variable number of frameblocks.
211 - 266	Transform all objects in the scene into screen space, using world-to-screen camera transformation.
278	Parameterize screen space as shown in Figure 7.
281 - 283	Use the Cohen-Sutherland algorithm to identify if an edge of an object is inside a frameblock.
285 - 287	Assign a color value to each object shader based on which frameblocks it resides in.

Table I: Description of shader setup code by line number. Source is shown in Appendix B.

Data was stored for each object inside of every frameblock, and exported for the frame in JSON format. Figure 9 shows an example of collecting distances of objects relative to a frameblock. We decided that since the distances of neighboring objects (the dashed lines) are unchanging for a static frame, it would be unproductive to include these values in our data.

Our semantics generation was successful for both the 8-bit and 16-bit precision models. We hope to use pairs of semantic data and final rendered output, to train an algorithm to produce the pixels of a frame given only its semantic data. Table I and Table II show an overview of the shader encoding and decoding scripts, respectively. Shader and Python source code is available in the Appendices of this document, while source files and final outputs are stored on GitHub [10].

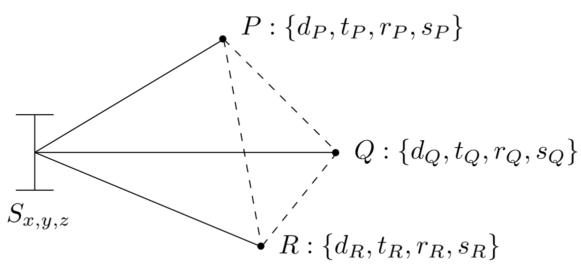


Figure 9: Relationships between points P, Q, and R and frameblock with world coordinates $S_{x,y,z}$.

V. Conclusion

Semantics data is easily extractable by humans – the human brain is easily able to recognize the relationships in scenes, while machines have a very hard time doing the same. Here we look at the possibility of generating semantics using data from the source, such that we are able to represent how a portion of a frame is rendered. Automating the process is a complicated task that involved consideration for scene dynamics and the relationship of objects to rendered data.

Line Number	Description
154 - 208	Collect color values of each object from their shaders.
210 - 228	Check if the calculated frameblocks of each object are equal to the ones it resides in.
230 - 241	Generate semantic data for all objects visible to each frameblock.
246 - 251	Export the collected semantic data in JSON format.

Table II: Description of semantics generation code by line number. Source is shown in Appendix C.

In the end, we successfully created a visually dynamic scene and utilized RenderMan shaders and Python scripting to generate semantics for each frameblock of the segmented screen. Our results show that a system for generating scene semantics per frame is feasible. After optimization, our implementation will be used to generate inputs for a machine learning application aimed at generating missing frames from scene semantics alone. We predict the research could be applied to solving difficult computer vision and other prevalent machine learning problems in Computer Graphics.

References

[1] J. Arvo, "Backward ray tracing," in *In ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, 1986, pp. 259–263.

[2] S. A. Pixel, "An overview of the ray-tracing rendering technique," https://www.scratchapixel.com/lessons/3d-basic-rendering/ray-tracing-overview, 2016.

[3] A. A. Apodaca and L. Gritz, *Advanced RenderMan: Creating CGI for Motion Picture*, 1st ed., B. A. Barsky, Ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999.

[4] I. Georgiev, T. Ize, M. Farnsworth, R. Montoya-Vozmediano, A. King, B. V. Lommel, A. Jimenez, O. Anson, S. Ogaki, E. Johnston, A. Herubel, D. Russell, F. Servant, and M. Fajardo, "Arnold: A brute-force production path tracer," *ACM Trans. Graph.*, vol. 37, no. 3, pp. 32:1–32:12, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3182160

[5] Pixar, "Renderman 20 documentation," https://renderman.pixar.com/resources/RenderMan_20/home.html, 2019.

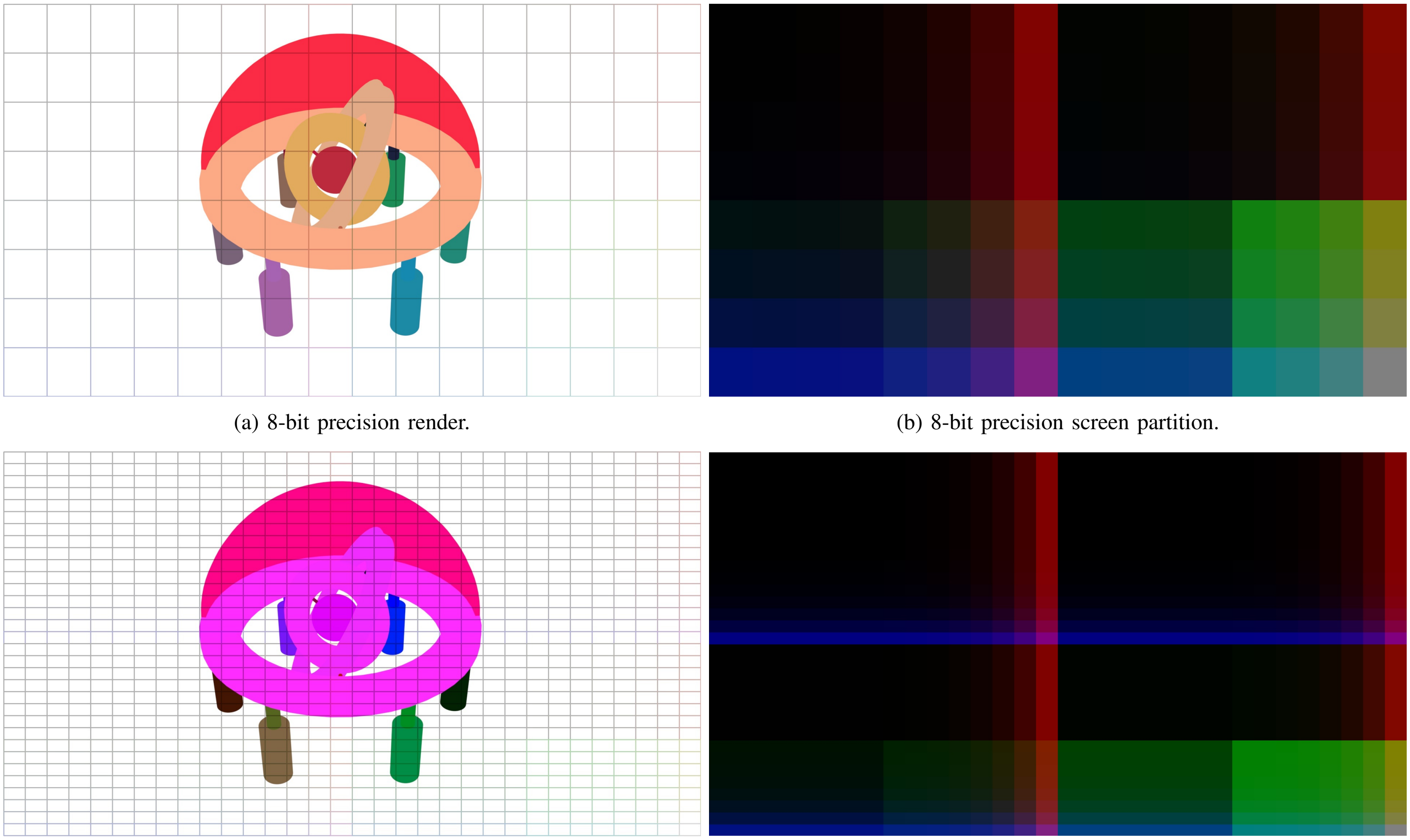
[6] P. Christensen, J. Fong, J. Shade, W. Wooten, B. Schubert, A. Kensler, S. Friedman, C. Kilpatrick, C. Ramshaw, M. Bannister, B. Rayner, J. Brouillat, and M. Liani, "Renderman: An advanced path-tracing architecture for movie rendering," *ACM Trans. Graph.*, vol. 37, no. 3, pp. 30:1–30:21, Aug. 2018. [Online]. Available: http://doi.acm.org/10.1145/3182162

[7] Pixar'sRenderMan, "Renderman 21: Quickstart," https://vimeo.com/pixarsrenderman, 2019.

[8] M. W. Harris, "Cs5800 demo 1," https://www.youtube.com/watch?v=UmGhuq_Zpr4, 2019.

[9] AcademicPhoenixPlus, "Intro to rigging in maya 2019," https://www.youtube.com/watch?v=1wvdQy2Fdhw, 2019.

[10] M. W. Harris, "Dynamic frame generation using machine learning and scene data," https://github.com/pixarninja/dynamic_frame_generator, 2018.



(a) 8-bit precision render.

(b) 8-bit precision screen partition.

(c) 16-bit precision render.

(d) 16-bit precision screen segmentation.

Figure 10: Renders of differing precisions compared to screen space segementation.

APPENDIX A

```
// programmable_rgb.osl
shader programmable_rgb(
int r = 0,
int g = 0,
int b = 0,
int n = 1,
output color resultRGB = color(0)
)
{
resultRGB = color((1.0 * r)/(pow(2.0, n) - 1),(1.0 * g)/(pow(2.0, n) - 1),(1.0 * b)/(pow(2.0, n) - 1));
}
```

APPENDIX B

```
# setup_shaders.py
import maya.cmds as cmds
import maya.OpenMaya as om
import maya.OpenMayaUI as omui
from functools import partial

##########################
#### Helper Functions ####
##########################

# Convert world space to screen space
# https://video.stackexchange.com/questions/23382/maya-python-worldspace-to-screenspace-coordinates
def worldSpaceToScreenSpace(worldPoint):
```

```python
    # Find the camera
    view = omui.M3dView.active3dView()
    cam = om.MDagPath()
    view.getCamera(cam)
    camPath = cam.fullPathName()

    # Get the dagPath to the camera shape node to get the world inverse matrix
    selList = om.MSelectionList()
    selList.add(cam)
    dagPath = om.MDagPath()
    selList.getDagPath(0,dagPath)
    dagPath.extendToShape()
    camInvMtx = dagPath.inclusiveMatrix().inverse()

    # Use a camera function set to get projection matrix, convert the MFloatMatrix
    # into a MMatrix for multiplication compatibility
    fnCam = om.MFnCamera(dagPath)
    mFloatMtx = fnCam.projectionMatrix()
    projMtx = om.MMatrix(mFloatMtx.matrix)

    # Multiply all together and do the normalisation
    mPoint = om.MPoint(worldPoint[0],worldPoint[1],worldPoint[2]) * camInvMtx * projMtx
    x = (mPoint[0] / mPoint[3] / 2 + .5)
    y = 1 - (mPoint[1] / mPoint[3] / 2 + .5)

    return [x,y]

# Collect all objects in the scene using Maya ls command
# https://stackoverflow.com/questions/22794533/maya-python-array-collecting
def collectObjects(currSel):
    meshSel = []
    for xform in currSel:
        shapes = cmds.listRelatives(xform, shapes=True) # it's possible to have more than one
        if shapes is not None:
            for s in shapes:
                if cmds.nodeType(s) == 'mesh':
                    meshSel.append(xform)

    return meshSel

# Test if the mesh is bounded by the coordinates
# https://boomrigs.com/blog/2016/1/12/how-to-get-mesh-vertex-position-through-maya-api
def testMesh(mesh, bounds):
    # Store bounds
    left = bounds[0][0]
    right = bounds[0][1]
    top = bounds[1][0]
    bottom = bounds[1][1]

    # Get Api MDagPath for object
    activeList = om.MSelectionList()
    activeList.add(mesh)
    dagPath = om.MDagPath()
    activeList.getDagPath(0, dagPath)

    # Iterate over all the mesh vertices and get position
    mItEdge = om.MItMeshEdge(dagPath)
    while not mItEdge.isDone():
        startPoint = mItEdge.point(0, om.MSpace.kWorld)
        endPoint = mItEdge.point(1, om.MSpace.kWorld)

        # Return with a True value if the edge is within the boundaries
        if clippingTest(startPoint, endPoint, bounds):
            return True

        mItEdge.next()

    return False

# Perform the Cohen-Sutherland Clipping test using Op Codes
# https://en.wikipedia.org/wiki/Cohen%E2%80%93Sutherland_algorithm
def clippingTest(p, q, bounds):
    P = worldSpaceToScreenSpace(p)
    Q = worldSpaceToScreenSpace(q)
```

```python
    opCodeP = opCode(P, bounds)
    opCodeQ = opCode(Q, bounds)

    # Trivial reject
    if (opCodeP & opCodeQ):
        return False

    return True

# Return the Op Code for a given point
def opCode(p, bounds):
    code = 0

    # Left of clipping window
    if p[0] < bounds[0][0]:
        code = code | 1

    # Right of clipping window
    if p[0] > bounds[0][1]:
        code = code | 2

    # Above clipping window
    if p[1] < bounds[1][0]:
        code = code | 4

    # Below clipping window
    if p[1] > bounds[1][1]:
        code = code | 8

    return code

# Update the color of a shader given r, g, b
def updateShaderColor(mesh, colorCode, n):
    shader = findShader(mesh)
    cmds.setAttr ( (shader) + '.r', colorCode[0] )
    cmds.setAttr ( (shader) + '.g', colorCode[1] )
    cmds.setAttr ( (shader) + '.b', colorCode[2] )
    cmds.setAttr ( (shader) + '.n', n )

# Return correct shader given a shader name
def findShader(mesh):
    cmds.select(mesh)
    nodes = cmds.ls(sl=True, dag=True, s=True)
    shadingEngine = cmds.listConnections(nodes, type='shadingEngine')
    materials = cmds.ls(cmds.listConnections(shadingEngine), materials=True)

    # Find the OSL shader node from connected nodes of the material
    for node in cmds.listConnections(materials):
        if node.find('PxrOSL') > -1:
            return node
    return None

##########################
### Main Functionality ###
##########################

# Create and display menu system
def displayWindow():
    menu = cmds.window( title="Setup Semantics Tool", iconName='SetupSemanticsTool', widthHeight=(350, 400) )
    scrollLayout = cmds.scrollLayout( verticalScrollBarThickness=16 )
    cmds.flowLayout( columnSpacing=10 )
    cmds.columnLayout( cat=('both', 25), rs=10, cw=340 )
    cmds.text( label="\nThis is the \"Semantics Shader Tool\"! This tool will generate semantics shaders for
               the loaded scene.\n\n", ww=True, al="left" )
    cmds.text( label="To run:\n1) Input the information in the fields below.\n2) Click \"Run\".", al="left" )
    cmds.text( label='Enter the keyframe at which to start semantics generation (1):', al='left', ww=True )
    startTimeField = cmds.textField()
    cmds.text( label='Enter the keyframe at which to end semantics generation (1):', al='left', ww=True )
    endTimeField = cmds.textField()
    cmds.text( label='Enter the step at which to process frames (1):', al='left', ww=True )
    stepTimeField = cmds.textField()
    cmds.text( label='Enter the number of bits used to store each, a multiple of 8 is recommended (8):',
               al='left', ww=True )
    bitNumField = cmds.textField()
```

```
    cmds.button( label='Run', command=partial( setupShaders, menu, startTimeField, endTimeField, stepTimeField,
                 bitNumField ) )
    cmds.text( label="\n", al='left' )
    cmds.showWindow( menu )

def setupShaders( menu, startTimeField, endTimeField, stepTimeField, bitNumField, *args ):
    # Grab user input and delete window
    startTime = cmds.textField(startTimeField, q=True, tx=True )
    if (startTime == ''):
        print 'WARNING: Default start time (1) used...'
        startTime = '1'
    endTime = cmds.textField(endTimeField, q=True, tx=True )
    if (endTime == ''):
        print 'WARNING: Default end time (1) used...'
        endTime = '1'
    stepTime = cmds.textField(stepTimeField, q=True, tx=True )
    if (stepTime == ''):
        print 'WARNING: Default step time (1) used...'
        stepTime = '1'
    bitNum = cmds.textField(bitNumField, q=True, tx=True )
    if (bitNum == ''):
        print 'WARNING: Default bit number (8) used...'
        bitNum = '8'
    N = int(bitNum)
    cmds.deleteUI( menu, window=True )

    # Set up program
    resWidth = cmds.getAttr('defaultResolution.width')
    resHeight = cmds.getAttr('defaultResolution.height')
    blockDim = [int(resWidth / (2 * N)), int(resHeight / ((N / 8) * N))]
    xDiv = float(resWidth) / blockDim[0]
    yDiv = float(resHeight) / blockDim[1]
    step = (resWidth / blockDim[0]) / (N / 2)

    # Set up blocks
    blocks = []
    for h in range(int(yDiv)):
        row = []

        # Find boundaries for each block in the row
        top = h / yDiv
        bottom = (h + 1) / yDiv
        for w in range(int(xDiv)):
            left = w / xDiv
            right = (w + 1) / xDiv

            row.append([[left,right],[top,bottom]])

        # Append the finished row
        blocks.append(row)

    print('Block Dim: (%d, %d), Blocks: (%d, %d)' % (blockDim[0], blockDim[1], len(blocks), len(blocks[0])))

    # Obtain all meshes in the scene
    currSel = cmds.ls()
    meshes = collectObjects(currSel)
    meshColors = []
    for n in range(len(meshes)):
        meshColors.append([0x0, 0x0, 0x0])

    # Iterate over all meshes and all boundaries
    for k, mesh in enumerate(meshes):
        cmds.select(mesh)
        bb = cmds.xform( q=True, bb=True, ws=True )

        # Obtain all 8 points to test from the bounding box
        # Format: xmin ymin zmin xmax ymax zmax
        bbPoints = []
        bbPoints.append(om.MPoint( bb[0], bb[1], bb[2], 1.0 ))
        bbPoints.append(om.MPoint( bb[0], bb[1], bb[5], 1.0 ))
        bbPoints.append(om.MPoint( bb[0], bb[4], bb[2], 1.0 ))
        bbPoints.append(om.MPoint( bb[0], bb[4], bb[5], 1.0 ))
        bbPoints.append(om.MPoint( bb[3], bb[1], bb[2], 1.0 ))
        bbPoints.append(om.MPoint( bb[3], bb[1], bb[5], 1.0 ))
```

```python
        bbPoints.append(om.MPoint( bb[3], bb[4], bb[2], 1.0 ))
        bbPoints.append(om.MPoint( bb[3], bb[4], bb[5], 1.0 ))

        # Translate to screen space and obtain overall bounds
        left, right, top, bottom = 1.0, 0.0, 1.0, 0.0
        for p in bbPoints:
            P = worldSpaceToScreenSpace(p)
            if left > P[0]:
                left = P[0]
            if right < P[0]:
                right = P[0]
            if top > P[1]:
                top = P[1]
            if bottom < P[1]:
                bottom = P[1]

        if left < 0.0 or left >= 1.0:
            left = 0.0
        if right > 1.0 or right <= 0.0:
            right = 1.0
        if top < 0.0 or top >= 1.0:
            top = 0.0
        if bottom > 1.0 or bottom <= 0.0:
            bottom = 1.0

        # Translate bounds to i and j values
        bounds = [int(left * len(blocks[0])), int(right * len(blocks[0])) + 1, int(top * len(blocks)),
                  int(bottom * len(blocks)) + 1]
        if bounds[0] > len(blocks[0]) - 1:
            bounds[0] = len(blocks[0]) - 1
        if bounds[1] > len(blocks[0]) - 1:
            bounds[1] = len(blocks[0]) - 1
        if bounds[2] > len(blocks) - 1:
            bounds[2] = len(blocks) - 1
        if bounds[3] > len(blocks) - 1:
            bounds[3] = len(blocks) - 1

        print('Processing {}: [({},{}),({},{})]'.format(mesh, bounds[0], bounds[1], bounds[2], bounds[3]))

        for i in range(bounds[2], bounds[3] + 1):
            b = i % N
            for j in range(bounds[0], bounds[1] + 1):
                r = j % N
                g = int((i / (N / 2))) * int(step) + int((j / (N / 2)))

                # Find bounds and color code for current block
                subBounds = blocks[i][j]
                colorCode = [0x1 << r, 0x1 << g, 0x1 << b]

                # Test which meshes are contained within the block
                if testMesh(mesh, subBounds):
                    for n in range(len(colorCode)):
                        meshColors[k][n] |= colorCode[n]

    for k, mesh in enumerate(meshes):
        updateShaderColor(mesh, meshColors[k], N)
        print(mesh, meshColors[k])

########################
####### Run Script #######
########################

# Display window
displayWindow()
```

APPENDIX C

```python
# extract_semantics.py
import maya.cmds as cmds
import maya.OpenMaya as om
import maya.OpenMayaUI as omui
from functools import partial
import json as json
import os as os

########################
#### Helper Functions ####
########################

# Convert screen space to world space
# https://forums.autodesk.com/t5/maya-programming/getting-click-position-in-world-coordinates/td-p/7578289
def screenSpaceToWorldSpace(screenPoint):
    worldPos = om.MPoint() # out variable
    worldDir = om.MVector() # out variable

    activeView = omui.M3dView().active3dView()
    activeView.viewToWorld(int(screenPoint[0]), int(screenPoint[1]), worldPos, worldDir)

    return worldPos

# Collect all objects in the scene using Maya ls command
# https://stackoverflow.com/questions/22794533/maya-python-array-collecting
def collectObjects(currSel):
    meshSel = []
    for xform in currSel:
        shapes = cmds.listRelatives(xform, shapes=True) # it's possible to have more than one
        if shapes is not None:
            for s in shapes:
                if cmds.nodeType(s) == 'mesh':
                    meshSel.append(xform)

    return meshSel

# Return the bit code for shader inputs and block offsets
def bitCode(mesh, r, g, b):
    shader = findShader(mesh)
    rVal = cmds.getAttr ( (shader) + '.r' )
    gVal = cmds.getAttr ( (shader) + '.g' )
    bVal = cmds.getAttr ( (shader) + '.b' )

    return [(rVal >> r) & 0x1, (gVal >> g) & 0x1, (bVal >> b) & 0x1]

# Test if the color value implies block intersection
def checkBitCode(code):
    if code[0] == 1 and code[1] == 1 and code[2] == 1:
        return True

    return False

# Return correct shader given a shader name
def findShader(mesh):
    cmds.select(mesh)
    nodes = cmds.ls(sl=True, dag=True, s=True)
    shadingEngine = cmds.listConnections(nodes, type='shadingEngine')
    materials = cmds.ls(cmds.listConnections(shadingEngine), materials=True)

    # Find the OSL shader node from connected nodes of the material
    for node in cmds.listConnections(materials):
        if node.find('PxrOSL') > -1:
            return node
    return None

# Extract semantic data based on block position and meshes in block
def extractSemantics(meshes, screenPoint, neighbors, cutoff):
    semantics = []

    for mesh in meshes:
        semanticsPerMesh = []
```

```
        for neighbor in neighbors:
            if mesh == neighbor:
                worldPoint = screenSpaceToWorldSpace(screenPoint)
                d = postionDistance(meshPosition(mesh), worldPoint)
                semanticsPerMesh.append('Screen : {}'.format( d ))
                continue

            d = findDistance(mesh, neighbor)
            if d <= cutoff:
                semanticsPerMesh.append('{} : {}'.format( neighbor, d ))

        semantics.append('[{} : {}]'.format( mesh, semanticsPerMesh ))

    return semantics

# Return the Euclidean distance between the centers of two meshes
def findDistance(meshA, meshB):
    return postionDistance(meshPosition(meshA), meshPosition(meshB))

# Obtain the position of a mesh in world space
def meshPosition(mesh):
    cmds.select(mesh)
    return cmds.xform( q=True, ws=True, t=True )

# Find the distance between two points
def postionDistance(posA, posB):
    return ((posA[0] - posB[0])**2 + (posA[1] - posB[1])**2 + (posA[2] - posB[2])**2)**0.5

##########################
### Main Functionality ###
##########################

# Create and display menu system
def displayWindow():
    menu = cmds.window( title="Extract Semantics Tool", iconName='ExtractSemanticsTool', widthHeight=(350,400) )
    scrollLayout = cmds.scrollLayout( verticalScrollBarThickness=16 )
    cmds.flowLayout( columnSpacing=10 )
    cmds.columnLayout( cat=('both', 25), rs=10, cw=340 )
    cmds.text( label="\nThis is the \"Extract Sematics Tool\"! This tool will extract semantics for the
               loaded scene.\n\n", ww=True, al="left" )
    cmds.text( label="To run:\n1) Input the information in the fields below.\n2) Click \"Run\".", al="left" )
    cmds.text( label='Enter the keyframe at which to start semantics generation (1):', al='left', ww=True )
    startTimeField = cmds.textField()
    cmds.text( label='Enter the keyframe at which to end semantics generation (1):', al='left', ww=True )
    endTimeField = cmds.textField()
    cmds.text( label='Enter the step at which to process frames (1):', al='left', ww=True )
    stepTimeField = cmds.textField()
    cmds.text( label='Enter the cut off distance for per-object semantics (100):', al='left', ww=True )
    cutOffField = cmds.textField()
    cmds.button( label='Run', command=partial( generateSemantics, menu, startTimeField, endTimeField,
                 stepTimeField, cutOffField ) )
    cmds.text( label="\n", al='left' )
    cmds.showWindow( menu )

def generateSemantics( menu, startTimeField, endTimeField, stepTimeField, cutOffField, *args ):
    # Grab user input and delete window
    startTime = cmds.textField(startTimeField, q=True, tx=True )
    if (startTime == ''):
        print 'WARNING: Default start time (1) used...'
        startTime = '1'
    endTime = cmds.textField(endTimeField, q=True, tx=True )
    if (endTime == ''):
        print 'WARNING: Default end time (1) used...'
        endTime = '1'
    stepTime = cmds.textField(stepTimeField, q=True, tx=True )
    if (stepTime == ''):
        print 'WARNING: Default step time (1) used...'
        stepTime = '1'
    cutOff = cmds.textField(cutOffField, q=True, tx=True )
    if (cutOff == ''):
        print 'WARNING: Default cutoff (100) used...'
        cutOff = '100'
    cmds.deleteUI( menu, window=True )
```

```
    # Set up program
    resWidth = cmds.getAttr('defaultResolution.width')
    resHeight = cmds.getAttr('defaultResolution.height')
    blockDim = 0 # Placeholder

    # Obtain all meshes in the scene
    currSel = cmds.ls()
    meshes = collectObjects(currSel)
    meshBlocks = {}

    # Iterate over all meshes
    xNum, yNum = None, None
    blocks = []
    blockToMeshMap = []
    for k, mesh in enumerate(meshes):
        shader = findShader(mesh)
        N = cmds.getAttr ( (shader) + '.n' )

        if blockDim == 0:
            blockDim = [int(resWidth / (2 * N)), int(resHeight / ((N / 8) * N))]

        xDiv = float(resWidth) / blockDim[0]
        yDiv = float(resHeight) / blockDim[1]
        step = (resWidth / blockDim[0]) / (N / 2)

        # Set up blocks
        if xNum is None or yNum is None:
            for h in range(int(yDiv)):
                row = []
                blockToMeshMap.append([])

                # Find boundaries for each block in the row
                top = h / yDiv
                bottom = (h + 1) / yDiv
                for w in range(int(xDiv)):
                    left = w / xDiv
                    right = (w + 1) / xDiv

                    row.append([[left,right],[top,bottom]])
                    blockToMeshMap[h].append([])

                # Append the finished row
                blocks.append(row)

            yNum = len(blocks)
            xNum = len(blocks[0])
            print('Block Dim: (%d, %d), Blocks: (%d, %d)' % (blockDim[0], blockDim[1],
                  len(blocks[0]), len(blocks)))

        # Iterate over all boundaries
        print('Evaluating {}...'.format( mesh ))
        for i in range(yNum):
            b = i % N
            for j in range(xNum):
                r = j % N
                g = int((i / (N / 2))) * int(step) + int((j / (N / 2)))

                # Check bit code of mesh for current block
                code = bitCode(mesh, r, g, b)
                if checkBitCode(code):
                    if mesh in meshBlocks:
                        meshBlocks[mesh].append([ r, g, b ])
                        blockToMeshMap[i][j].append(mesh)
                    else:
                        meshBlocks[mesh] = [[ r, g, b ]]
                        blockToMeshMap[i][j] = [mesh]

    # Check if the algorithm correctly extracted the blocks
    for k, mesh in enumerate(meshes):
        meshColors = [0x0, 0x0, 0x0]
        if mesh in meshBlocks:
            for c in meshBlocks[mesh]:
                colorCode = [0x1 << c[0], 0x1 << c[1], 0x1 << c[2]]
```

```
                for n in range(len(colorCode)):
                    meshColors[n] |= colorCode[n]
        else:
            print('{}: No blocks found!'.format( mesh ))

        shader = findShader(mesh)
        rVal = cmds.getAttr ( (shader) + '.r' )
        gVal = cmds.getAttr ( (shader) + '.g' )
        bVal = cmds.getAttr ( (shader) + '.b' )
        if (meshColors[0] == rVal) and (meshColors[1] == gVal) and (meshColors[2] == bVal):
            print('{}: Good!'.format( mesh ))
        else:
            print('{}: {} ({},{},{})'.format( mesh, meshColors, rVal, gVal, bVal ))

    # Extract semantics for each mesh
    semantics = []
    for i, data in enumerate(blockToMeshMap):
        row = []
        for j, meshesInBlock in enumerate(data):
            if not meshesInBlock:
                print('No semantics for block({},{})'.format( i, j ))
            else:
                # Screen point = Ydim * (i + 1), Xdim * (j + 1)
                screenPoint = blockDim[1] * (i + 0.5), blockDim[0] * (j + 0.5)
                row.append('({}, {}) : {}'.format( i, j, extractSemantics(meshesInBlock, screenPoint, meshes,
                           float(cutOff)) ))
        semantics.append(row)

    for row in semantics:
        print(row)

    # Write data to an output file
    filepath = cmds.file(q=True, sn=True)
    filename = os.path.basename(filepath)
    raw_name, extension = os.path.splitext(filename)
    with open('C:\\Users\\wesha\\Documents\\maya\\projects\\CS5800\\scenes\\{}_output_{}.txt'.format
              ( raw_name, N ), 'w') as f:
        f.write( json.dumps(semantics).replace('"', '').replace('\'', '') )

##########################
####### Run Script #######
##########################

# Display window
displayWindow()
```