SceneScript: Constraint-Based Scene Definition

Hanna Winter
hannawii@stanford.edu

Shannon Kao
kaos@stanford.edu

ABSTRACT

SceneScript is a web-based 3D scene definition language. Explicit scene definition formats are difficult to use and iterate over. SceneScript uses constraints to encode relational data into object placement, making scene layout a more efficient, natural task.

1. INTRODUCTION

Laying out a scene can be a slow and painful task, especially as the number of objects in the scene increases. In traditional modelling software, a user must manually determine the precise translation, rotation, and size of each object. The simple real-world task of placing multiple bowls on a table, then, becomes the much more arduous job of precisely aligning the base of each bowl with the surface of the table. And where modelling software's direct manipulation interface is somewhat intuitive, current web-based 3D scene formats offer even fewer affordances, often requiring users to enter numeric values to position each object in their scene.

It is natural to model the world as a set of relations between objects, e.g. "the bowl is *on* the table". Explicit scene definition formats are unintuitive in that they require users to translate this mental model into numeric 3D coordinates. These formats are also rigid–to generate variations on one scene or to iterate over placements requires every object in the scene to be manually repositioned.

We introduce SceneScript, a constraint-based scene definition format in JavaScript. SceneScript introduces constraints to define the relationship between two objects. These constraints allow users to specify object positions relative to existing scene geometry. We also introduce the idea of a range, a specialized constraint that can be applied to a single property. SceneScript is a modular system that can be easily connected to existing JavaScript-based rendering engines, allowing users to quickly and intuitively specify complex 3D scene layouts without restructuring their existing workflow.

2. OVERVIEW

SceneScript is a compact language designed to intuitively lay out a 3D scene. In the following section we will introduce the language from a user's perspective and describe how the system interprets and processes the language.

2.1 Language

SceneScript is embedded in JavaScript, allowing the user to seamlessly integrate our system into an existing pipeline. The basic format of the language is designed to adhere to intuitive, real-world principles of scene layout.

2.1.1 Objects

Users first initialize scene objects, consisting of a set of properties and a geometry. This geometry is defined by a string that is either the URL of an OBJ file or the name of a built-in shape (e.g. "sphere", "cube"). The properties may be defined as numeric values or as a range (max, min), which will be sampled when the scene is evaluated.

Translate, Rotate, Scale a set of three parameters x, y, and z, defining the position, orientation, and size of the object. These are the only properties that can be constrained.

Material a hex code or the url of a texture, to be applied to the object

Bounding Box two vectors min and max each with x, y and z parameters, defining the bounding box. This is a convenience property, calculated from the scale, rotate, and translate of the object. It is not set by the user.

2.1.2 Constraints

After defining objects, users then place this geometry in their scene by applying constraints. Objects may have numeric properties defined on creation, but these will be overwritten if constraints are present, as demonstrated in Figure 1. A constraint is a function that calculates one property of an object based on already-calculated values from a different property. Constraints encode a relationship between two pieces of geometry, allowing for rapid iteration on positioning, as well as logical variations on a scene.

2.1.3 World

Each scene defined in SceneScript must have a world object, which acts as the parent of the entire scene. All objects are

```
y_constraint = function( A, B ) {
     A.y = B.y * 2;
}
bowl.addConstraint( "translate",
                    table.translate,
                    y_constraint );
```

Figure 1: A constraint is a function applied to one property of an object. Here, the bowl's y-translation is constrained by the table's y-translation. The user defines a specific constraint function, multiplying the table's y-position by two.

added to the world, and the user simply calls `world.eval()` in order to generate a specific instance of their scene.

2.2 System

Internally, SceneScript constructs a directed, acyclic graph to represent the scene as the user adds objects and constraints to the world. When the user calls `world.eval()`, the system performs a depth-first traversal of this graph in order to generate numeric values for each object property.

2.2.1 Graph

When a new object is added to the scene, the system adds another node to this graph. It loops over each constraint applied to the object and adds an edge pointing from the parent of the constraint to the new object.

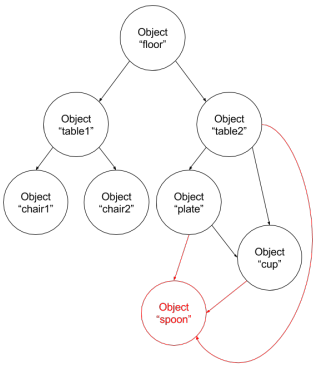


Figure 2: A new node ("spoon") is added to the scene graph. The spoon's placement is constrained by the table, the plate, and the cup, so these constraint edges are added to the graph.

This graph is modified each time an additional constraint is added to the scene as well, so that it accurately reflects the scene at any given point in time.

2.2.2 Traversal

Fundamentally, scene generation is a depth-first traversal of the scene graph. One-way constraints are required, so any node in the graph is only dependant on values in its parent nodes. Traversing the graph in this way ensures that no property attempts to access a value that has not yet been generated.

The system first checks the graph to find root nodes–that is, unconstrained nodes. If a root node is not found, an error is returned. During evaluation, each object stores a temporary vector of numeric values. On evaluation, ranges are sampled, existing numeric values are copied over, and constraints are then applied to the values. These values are then translated to the JSON output format (Figure 3).

```
constraints:
    translate.z = table.translate.z * 2;

 translate:          translate:   translate:
 x:  2               x:  2        x:  2
 y:  range(1, 10)    y:  7.45     y:  7.45
 z:  1               z:  1        z:  4 * 2
```

Figure 3: Unevaluated constraints are stored in our system. On scene traversal, ranges are first sampled as in translate.y, then constraints are applied, overwriting previous values where necessary as seen in translate.z.

Cycles SceneScript currently does not allow cycles in the scene graph. If, during the traversal, the system encounters a node that has already been fully processed, an error is returned. While certain configurations of cyclic graphs may be valid scenes, the majority are not. Additionally, due to the hierarchical nature of real-world scenes (often simply a room, with large objects in the room, and smaller objects placed on the large objects), we find that users rarely need cyclic constraints to properly denote their layouts.

2.3 Output

The output of SceneScript is formatted JSON, with numeric values for each property.


```
"{
    "floor":{
        "translate":[0,2,0.47892178761898996],
        "rotate":[-1.2,0,1.4845559450841874],
        "scale":[10,10,1],
        "count":0,
        "geometry":"plane",
        "material":"textures/wood_floor_texture.jpg"
    },"table":{
        "translate":[2.3202286343166687,2,0.11980392356955627],
        "rotate":[0.370795,1.4845559450841874,0],
        "scale":[1,1,1],
        "count":0,
        "geometry":"obj/table.obj",
        "material":"0x492116"
    },"chair1":{
        "translate":[2.3202286343166687,2,0.11980392356955627],
        "rotate":[0.370795,1.4845559450841874,0],
        "scale":[1,1,1],
        "count":0,
        "geometry":"obj/chair.obj",
        "material":"0xA64F2F"
    },"chair2":{
        "translate":[2.3202286343166687,2,0.11980392356955627],
        "rotate":[0.370795,4.626145945084187,0],
        "scale":[1,1,1],
        "count":0,
        "geometry":"obj/chair.obj",
        "material":"0xA64F2F"
    },"cup":{
        "translate":[2.3202286343166687,3.9699999999999998,0.11980392356955627],
        "rotate":[0.370795,1.4845559450841874,0],
        "scale":[0.021,0.021,0.021],
        "count":0,
        "geometry":"obj/cup.obj",
        "material":"textures/text.jpg"
    },"lamp":{
        "translate":[1.6202286343166687,3.1,1.9198039235695563],
        "rotate":[0.370795,1.4845559450841874,0],
        "scale":[0.01,0.01,0.01],
        "count":0,
        "geometry":"obj/table_lamp.obj",
        "material":"0x121212"
    }
}"
```


Figure 4: SceneScript output for a test scene with six objects.

To translate the scene into a renderable 3D format, a straightforward loop over each object in the JSON output will suf-

fice. We implemented a simple parser to convert this JSON to a THREE.js scene with appropriate lighting for our results.

3. EVALUATION

We ran quantitative performance tests to evaluate SceneScript, in addition qualitative analysis of user workflow while working with our language.

3.1 Test Scene

We implemented a small demo to showcase the capabilities of SceneScript. Our demo initializes six objects.

- The floor plane is the root of the scene. The x- and y-translations are constant and the z-translation is `range(0, 10)`. The z-rotation is `range(0.5, 1.5)`.
- The table is a child of the floor, where the x-, y-, and z-translations are constrained to the floor translation and the x- and z-rotations are constrained to the floor rotation, as seen in Figure 5. The table geometry is defined by an OBJ url.
- Four additional objects consisting of two chairs, one cup, and one lamp geometry are added as children of the table.

```
$( document ).ready(function() {
    // Create world object
    var w = new World();

    // Create floor object, passing in unique id ("floor")
    var floor = new ProceduralObject('floor');

    // Floor properties
    floor.setTranslate(0,2,new Range(0,10));
    floor.setRotate(-1.2,0,new Range(0.5, 1.5));
    floor.setScale(10,10,1);
    floor.setMaterial("textures/wood_floor_texture.jpg");
    floor.setGeometry('plane');

    // Add the floor object to the world
    w.addObject(floor);

    // Create table object, passing in unique id ("table")
    var table = new ProceduralObject('table');

    // Table properties
    table.setGeometry('obj/table.obj');
    table.setMaterial("0x492116");
    table.setScale(1,1,1);

    // Table constraint functions
    var table_translate = function(tableT, floorT) {
        var x_range = new Range(floorT.x-((floor.getScale().x)/2)+2,
                                floorT.x+((floor.getScale().x)/2)-2);
        tableT.x = tableT.prototype.sampleValue(x_range);

        tableT.y = floorT.y;

        var z_range = new Range(floorT.z-((floor.getScale().z)/2)+2,
                                floorT.z+((floor.getScale().z)/2)-2);
        tableT.z = tableT.prototype.sampleValue(z_range);
    }

    var table_rotate = function(tableR, floorR) {
        tableR.x = PI/2 + floorR.x;
        tableR.y = floorR.z;
    }

    // Add the table constraints on the translate of the floor and
    // rotation of the floor using the constraint functions as arguments
    table.addConstraint("translate", floor.getTranslate(), table_translate);
    table.addConstraint("rotate", floor.getRotate(), table_rotate);

    // Add the table object to the world
    w.addObject(table);
```

Figure 5: Code for placing a floor object and a table object in the world. Notice the table_translate and table_rotate functions that describe the constraints on the table's translate property and rotate property respectively.

Several rendered scenes (Figure 6) show that despite variation in floor translation and rotation, all the objects maintain the set constraints and user-specified relationships.

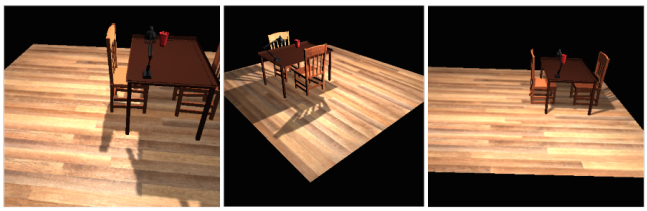


Figure 6: Three different renderings of our demo scene. The floor adjusts from image to image because of the z-translation defined by range(0,10) and the z-rotation defined by range(0.5, 1.5). The table's position varies with each rendering but remains on the floor as dictated by the constraints on the table's x-, y-, and z-translations and x- and z-rotations. The chairs, lamp, and cup remain constrained to the table despite variation in positioning.

3.2 Performance

We ran initial performance tests on a naive depth-first traversal over the graph, as defined in Section 2.2.2. The number of objects on the table, and corresponding constraints, were incremented for each test. In general, our system is fairly efficient on large numbers of objects and constraints, taking only 0.8s (Figure 7) to generate a 10,000 object scene. With our demo, the bottleneck was by far the rendering portion, handled by THREE.js.

However, the naive DFS visits the same node multiple times in order to calculate each constraint. That is, a table that is constrained to the floor's rotation and translation is visited twice, once through each constraint edge. To further optimize our system, we collapse these edges into one visit, calculating all constraints between two objects at once. This offered a significant speed-up, reducing the 10,000 object scene generation by nearly 50%.

	Unoptimized	**Edge Collapsing**
10 objects	0.25ms	0.2ms
100 objects	6ms	3.5ms
10000 objects	800ms	400ms

Figure 7: Evaluation time (ms) for scenes with 10, 100, and 10000 objects, respectively.

3.3 Workflow

Laying out our test scene in THREE.js requires the user to manually calculate values for the chairs, the cup, and the lamp in order to maintain the proper spatial relationships.

To adjust the scene, users must adjust the position of each object. With SceneScript, the user is able to adjust the table's position without having to modify the any properties of the cup, chairs, and lamp. Additionally, a simple refresh of the page generates variations of the scene based on the pre-defined range.

The JSON output allows programs written in SceneScript to be versatile and usable in many different renderers. Scene data can be precomputed or repeatedly recomputed on the fly to generate nearly infinite versions of the world.

4. DISCUSSION

In this section, we discuss some limitations of SceneScript and future work.

4.1 Acyclic Graphs

One of the major limitations of SceneScript is the simplified graph representation. Our traversal does not allow cycles, which reduces the variety of scenes that can be produced by SceneScript. Arbitrary constraint satisfaction is an extremely difficult problem–provably NP-complete over certain domains–so we have chosen to reduce our problem space to the subset of directed, acyclic graphs.

However, we could feasibly alter our graph structure to treat individual properties, rather than entire objects, as graph-nodes. Then, a scene where a table and a chair have a circular dependency would be permitted (Figure 8).

Additionally, we can further advance SceneScript by allowing users to have one object property depend on multiple different other object properties, including its own. With this adjusted graph structure, allowing one object property value to depend on two different object properties should have the same effect as our current graph implementation, shown in Figure 3, where two nodes, that are not necessarily siblings, can share a child.

4.2 Renderer Integration

JSON output allows information to be precomputed and used with different renderers. However, it then becomes the user's responsibility to parse this output into renderable data. Further, our format does not fully specify a scene, omitting objects like lights and cameras. One immediate way to better integrate SceneScript with a renderer would be to create a procedural representation of these objects, a fairly straightforward task in our current model.

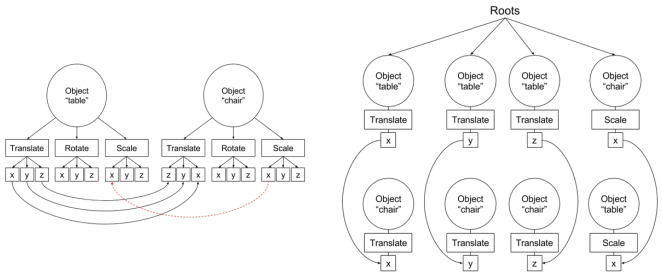
Another extension could be a language that stitches SceneScript to a renderer of choice. This language would parse the JSON output from a SceneScript program and render a scene reflecting the output.

4.3 Semantics

Currently, SceneScript relies on the user to write the constraint calculation function. This requires a basic knowledge of programming, making SceneScript daunting and inaccessible to certain populations. Experimentation with high level semantics could make SceneScript much more intuitive. Constraints like "on", "beside", "under", or "inside" could be used to describe object positioning, while keeping the current method of constraint specification intact for more advanced users. These syntactic shortcuts would generate the correct translation, rotation, and scale attributes behind the scenes to fulfill the intent of the constraint.

4.4 Physics-based Constraints

While SceneScript does encode naive randomness in the range property, this randomness does not always generate realistic parameters. Objects may interpenetrate, or be placed in orientations that are not physically plausible. A physics-based constraint would allow users to more easily create realistic scenes. In addition the range property, a physically-based constraint would run a physics simulation, or an approximation of one, in order to generate appropriate object positioning. For example, a gravity constraint might take the user-specified translation of a pen, and apply gravity until the pen hit a supporting surface.



(a) Graph A

(b) Graph B

Figure 8: Graph A depicts our current implementation where cycles are not allowed. The table's x-scale value cannot be dependent on the chair's x-scale value because the chair's translate property is already dependent on the table's translate property. Graph B represents the same information as A but with nodes as object properties rather than the objects themselves.

5. CONCLUSION

SceneScript is an efficient scene specification format that integrates smoothly with existing JavaScript pipelines. By allowing users to write constraint-based scenes, SceneScript makes laying out geometry a natural and intuitive process. Generating variations of a scene becomes trivial when object relationships are encoded into the scene structure by default. Additionally, the concept of ranges allows users to easily integrate randomness into a world. There is an exciting range of future steps, from syntactic sugar to physics-based constraints, that will make SceneScript more powerful and comprehensive.