A Scene Graph Framework

Fundamentals

At the heart of a 3D graphics framework is a **scene graph**: a data structure that organizes the contents of a 3D scene using a hierarchical or tree-like structure.

A **tree** represents a hierarchical nature of structure in a graphical form. It consists of elements called **nodes**, with each node links to its successors. The successors of a node are its **child nodes**, while the predecessor of a node is called its **parent**. Each child node has exactly one (1) parent except for a special node called the **root node**. This node is the tree's topmost node.

The following is an example of a tree.

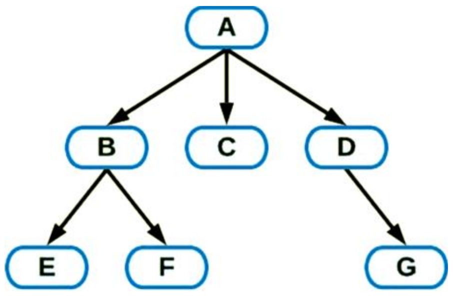

Figure 1*.* A tree with seven nodes.

- Nodes: A, B, C, D, E, F, G
- Root node: A
- Parent nodes: A, B, D
- Child nodes: B-C-D for A, E-F for B, and G for D

In a scene graph, each node represents a 3D object in the scene. The accumulated transformations applied to an object are stored as the product of the corresponding matrices, which is a single matrix called the **model matrix** of the object. The model matrix effectively stores the current location, orientation, and scale of an object. These three (3) are collectively referred to as the **transform** of the object. The model matrix stores the transform of an object relative to its parent object in the scene graph. The transform of an object relative to the root of the scene graph, often called a **world transformation**, can be calculated from the product of the model matrix of the object and those of each of its ancestors. This structure enables complicated transformations to be expressed in terms of simpler ones.

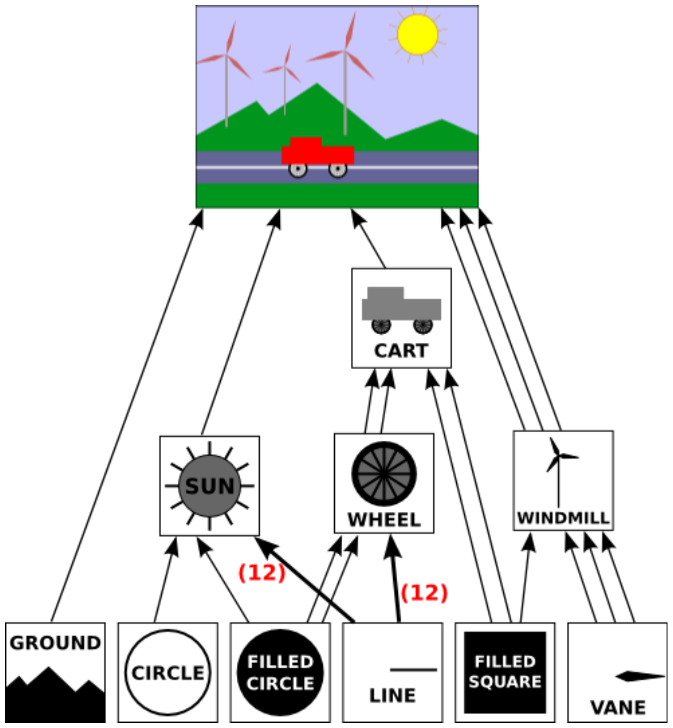

Figure 2*.* A scene and its scene graph.

In a scene graph, a single object can have several connections to one or more parent objects. Each connection represents an occurrence of the object in its parent object. For example, the filled square object is as a child object in the cart and the windmill. It is used twice in the cart and once in the windmill. The filled circle is used in the sun once and twice in the wheel. The line is used 12 times in the sun and also in the wheel. The thick arrows, marked with 12, represent 12 connections. The wheel is used twice in the cart.

Each arrow in the figure can be associated with a modeling transformation that places the child object into its parent object. When an object contains multiple copies of a child object, each arrow connecting the child object to the object will have a different associated modeling transformation. The object is the same for each copy, while the transformation differs.

A scene graph structure also allows simple geometric shapes to be grouped together into a compound object that can be easily transformed into a single unit. For example, a simple model of a table may be created using a large, flat box for the top surface and four (4) narrow, tall boxes positioned underneath near the corners for the table legs. Each of these objects can be stored in a node, and all five (5) nodes share the same parent node.

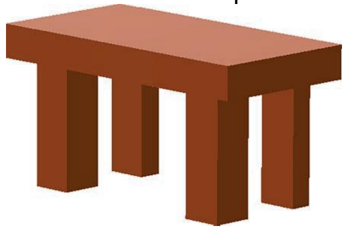

Figure 3*.* A table consisting of five boxes.

Transforming the parent node affects the entire table object. Note that each of these boxes may reference the same vertex data; the different sizes and positions of each may be set with a model matrix. A scene graph-based framework allows you to create interactive scenes containing complex objects, such as the one in the same sample figure below.

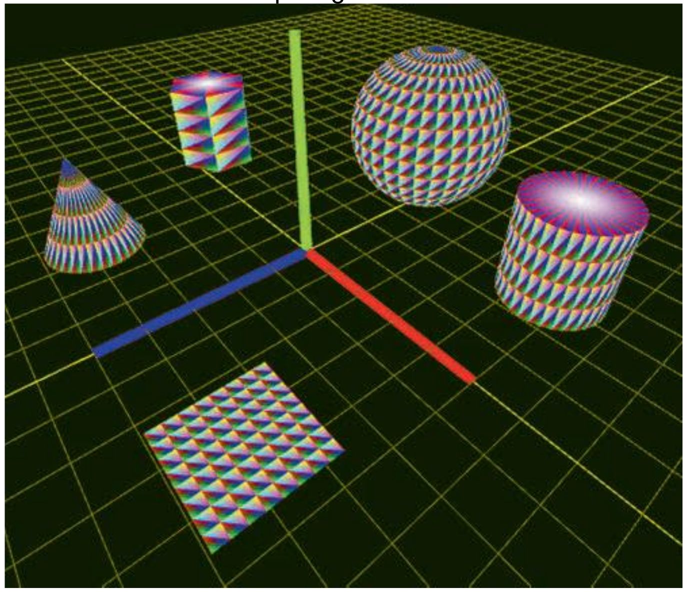

Figure 4*.* A scene containing multiple geometric shapes.

The Class Structure

In a scene graph framework, the nodes represent objects located in a three-dimensional space. The corresponding class (ex. `Object3D` class) shall contain three (3) items:

- A matrix to store its transform data
- A list of references to child objects
- A reference to a parent object

Many classes can extend the above class, each with a different role in the framework. These classes represent the following. (*Those in parentheses are sample class names.)*

1. Root node (`Scene`)
2. Interior nodes used for grouping purposes (`Group`)
3. Nodes corresponding to objects that can be rendered (`Mesh`)
4. Virtual camera for rendering the point of view (`Camera`)
5. Virtual light source that affects the shading and shadows (`Light`)
6. The general shape and other vertex-related properties of each mesh (`Geometry`)
7. The general appearance of an object (`Material`)
8. The general initialization tasks and image rendering (`Renderer`)

Since each instance of a mesh stores a transformation matrix, multiple versions of a mesh (based on the same geometry and material data) can be rendered with different positions and orientations. Each mesh shall store a reference to a **vertex array object (VAO)**. A VAO associates *vertex buffers* (whose references shall be stored by attribute objects stored in the geometry) to attribute variables (specified by shaders stored in the material). This will allow geometric objects to be reused and rendered with different materials in different meshes.

The class representing the vertex-related properties of each mesh (ex. `Geometry` class shall store `Attribute` objects that describe vertex properties, such as position and color. Geometric objects shall also store texture coordinates, for applying images to shapes, and normal vectors, for use in lighting calculations. This class shall calculate the total number of vertices equal to the length of the data array stored in any attribute. Classes extending the `Geometry` class shall be created to realize each particular shape. In some cases, such as rectangles and boxes, the attribute data shall be listed directly.

For other shapes, such as polygons, cylinders, and spheres, the attribute data shall be calculated from mathematical formulas.

The class representing the general appearance of an object (ex. `Material` class) shall serve as a repository for three (3) types of information related to the rendering process and the appearance of an object. These include the shader code (and the associated program reference), Uniform objects, and render settings. The render settings are the properties which are set by calling functions, such as the type of geometric primitives (points, lines, or triangles), point size, line width, and so forth. The base `Material` class shall initialize dictionaries to store uniform objects and render setting data. It shall also define functions to perform tasks such as compiling the shader program code and locating uniform variable references. Extensions of this class shall supply the actual shader code, a collection of uniform objects corresponding to uniform variables defined in the shaders, and a collection of render setting variables applicable to the type of geometric primitive being rendered.

The rendering function in the `Renderer` class shall require two (2) parameters: a scene object and a camera object. For each mesh in the scene graph, the renderer shall perform the tasks necessary before the function for drawing is called. These tasks include activating the correct shader program, binding a vertex array object, configuring render settings, and sending values to be used in uniform variables. There are three (3) required uniform variables by most shaders whose values are stored outside the material. These are the transformation of a mesh, the transformation of the virtual camera used for viewing the scene, and the perspective transformation applied to all objects in the scene. Though the uniform objects will be stored in the material for consistency, this matrix data shall be copied by the rendered into the corresponding uniform objects before they send their values to the GPU.

References:
Eck, D. (2021). Citing sources. Retrieved from https://math.hws.edu/
Korites, B. (2018). *Python graphics: A reference for creating 2D and 3D images.* Apress.
Marschner, S. & Shirley, P. (2021). *Fundamentals of computer graphics* (5th ed.). CRC Press.
Stemkoski, L. & Pascale, M. (2021). *Developing graphics frameworks with Python and OpenGL.* CRC Press.