Universal Relational Storage for Geometric Primitives

Timothy M. Shead*
The K-3D Project

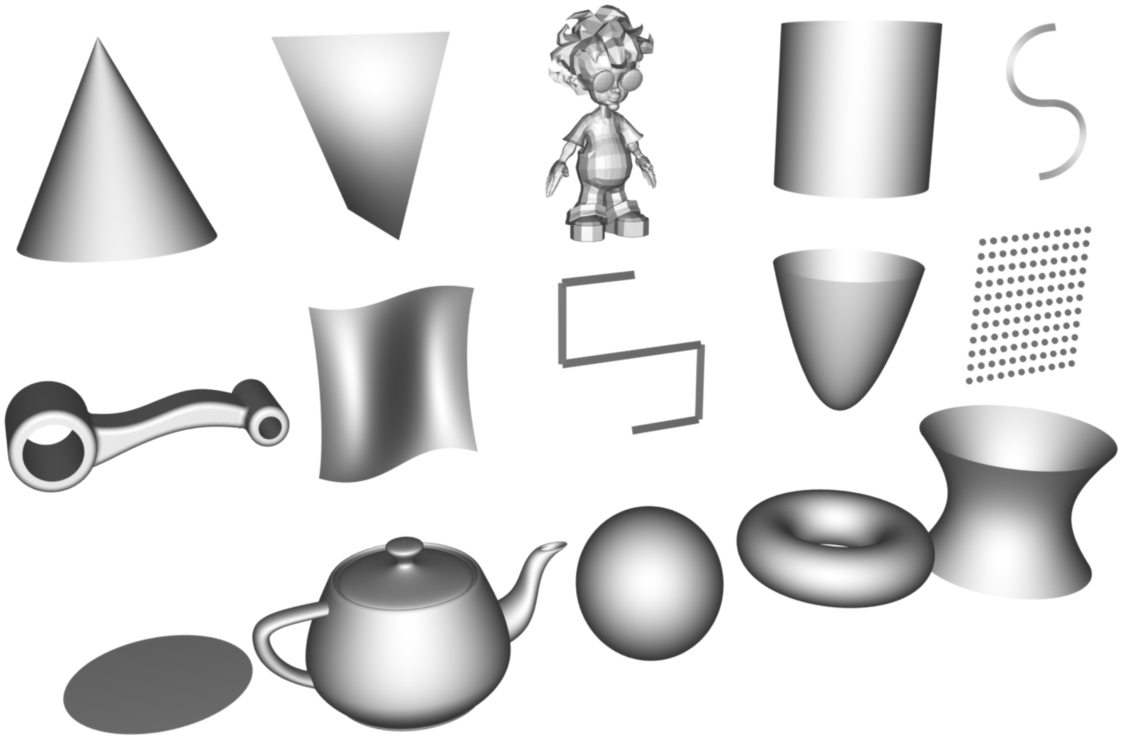

Figure 1: A zoo of geometric primitive types can be stored using a single relational data structure.

Abstract

To make new geometric primitive types easier to work with for researchers and easier to use for application developers, we propose a universal data structure that efficiently stores complex heterogeneous collections of geometric primitives (Figure 1). Primitives are stored using a relational hierarchy of strongly-typed named arrays that is more efficient than pointer-based structures, suitable for partitioning and parallel computation, and flexible enough to allow practical, “on-the-fly” generation of new primitive types at runtime. The arrays and relationships used to encode primitive types are captured using “schemas” inspired by relational database systems — making it possible to implement automated validation and modification operations without a priori understanding of how individual arrays will be used by a primitive. We demonstrate an implementation of this data structure within the visualization pipeline of the open-source K-3D modeling and animation application, and provide examples of builtin and runtime schemas.

CR Categories: I.3.5 [Computational Geometry and Object Modeling]; I.3.6 [Methodology and Techniques]: Graphics data structures and data types; E.1 [Data Structures]: Arrays geometric primitives, data structures, arrays

*e-mail: tshead@k-3d.com

1 Introduction

The set of geometric primitive types used in computer graphics is constantly growing: in addition to common types including polyhedra, NURBS curves and surfaces, and quadrics, new research into techniques including point-based surfaces[Kobbelt and Botsch 2004], implicit surfaces [Bloomenthal and Wyvill 1990], and subdivision surfaces [Zorin 2006] is ongoing. However, researchers working with new primitive types cannot use existing applications and are forced to spend time on application development and supporting-code instead of focusing on the primitives themselves. For applications, the ever-expanding list of new primitive variations and completely new primitive types is an embarrassment of riches. Providing support for the union of interesting geometric primitive types to end-users becomes increasingly difficult as adding hard-coded internal data structures for each new type requires significant amounts of repetitive, detail-oriented, and error-prone code to support serialization, validation, scripting, primitive-specific algorithms, etc. For both communities — researchers and application developers — something new is needed: a universal data structure that can be used to encode any primitive type, allowing for the rapid development, deployment and use of new primitives as the field advances. Such a data structure must balance a broad set of potentially-conflicting requirements: in addition to capturing the structure and attributes of widely-divergent primitive types, it must do so in a way that is efficient in time and space, supports modern CPU design and the trend towards parallel multi-threaded computation, and supports the features of modern graphics hardware such as retained geometry. The design must maintain a balance between generality and usability, and should be sufficiently self-describing to allow automated operations on otherwise unknown primitive types. Last-but-not-least, such a structure must provide a consumable programming interface and a reasonable match for developers' mental models. In the remainder of this paper we propose a data structure that meets these requirements, and describe a reference implementation using the open-source K-3D modeling and animation system [Shead 2005]. K-3D is based on a pipeline architecture where data source, data modifier, and data sink components are instantiated from plugins and connected to create data-processing pipelines [Shead 2008]. The flexibility of a universal data structure for primitives is uniquely suited to such a combined plugin / pipeline architecture, making it easy to create new plugin components that generate, modify, and render new primitive types without altering the core application. Using programmable pipeline components, it becomes practical to introduce new geometric primitive types in a running application using interpreted scripting languages — a unique and powerful feature for researchers and power-users. Note here the important distinction that we are making between creating a new primitive types (e.g. a "sphere" primitive) versus creating primitive instances (a polyhedral approximation to a sphere, a sphere made-up of NURBS patches, etc).

2 Design

2.1 Geometry

Our goal was to create a generic primitive data structure that would fit well within a visualization pipeline architecture, as in Figure 2. Here, a top-level `Geometry`data structure containing geometric primitives is created by a data source, travels through the pipeline to a data modifier where it is altered, then moves to a data sink where it is rendered or serialized:

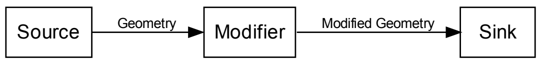

Figure 2: Visualization pipeline processing geometric data.

In many applications, a typical representation for geometry might be a C struct, C++, or Java class representing a hard-coded collection of specific primitive types (Figure 3):

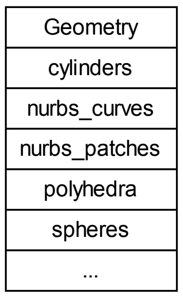

Figure 3: Typical geometry object containing hard-coded primitives.

Because we wanted to support an open-ended set of primitive types, we opted to begin our new `Geometry` data structure as a container for zero-to-many `Primitive` objects (Figure 4):

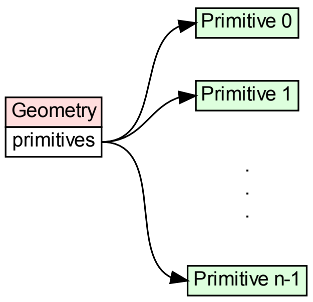

Figure 4: Our `Geometry` object is a container for generic `Primitive` objects.

Each `Primitive` contained in the top-level `Geometry` data structure encapsulates an instance of a specific geometric primitive type, and the `Geometry` can contain zero-to-many instances of any primitive. This approach adds significantly to the flexibility of the structure (we can move arbitrary numbers of arbitrary primitive types through the pipeline as a single logical unit) and greatly simplifies many operations, as will be seen later.

2.2 Primitive Storage

The most serious challenge in designing the `Primitive` data structure is striking the right balance between structure and generality. Typical pointer-based data structures encode primitives using constructs that are defined at compile-time, such as C++ classes. This data is highly structured (providing type-safety, validation, and constraint enforcement) but exhibits poor flexibility (adding a new type requires modifications to existing classes). At the opposite extreme, we could simply declare a primitive to be "a container of bytes" — an approach providing complete flexibility at the expense of being devoid of all structure, and consequently completely opaque to third parties. Since we wanted to allow for generic operations that could be be applied to any primitive, without prior understanding of its structure, some middle-ground was necessary.

For inspiration, we turned to the scientific simulation and visualization community, where it is common to represent finite element meshes using arrays of points, cells, and connectivity data. For example, the Visualization Toolkit (VTK) defines more than 15 cell-types including pixels, voxels, triangles, polygons, and linear and quadratic hexahedra, all stored using arrays [Schroeder et al. 2006]. Similarly, [Alumbaugh and Jiao 2005] present several schemes for storing polyhedra using collections of arrays, and enumerate the advantages of arrays over pointer-based data structures, including compact storage, comprehensive support in traditional programming languages, convenient data exchange among libraries and applications, and easy partitioning for parallel computation. They also outline strategies for efficient querying of entities and neighborhoods. Finally, the RIB binding to the RenderMan interface provides a compelling real-world example of how different primitive types can be encoded using varying-length arrays[Pixar 2005]. These examples convinced us that it was possible, though not sufficient, to encode arbitrary primitive types using collections of arrays.

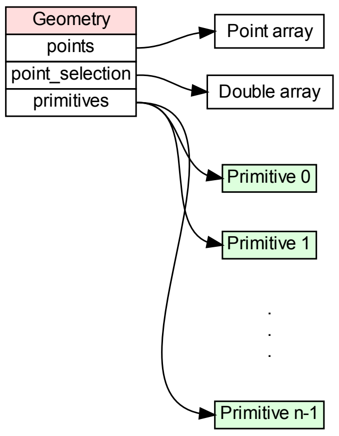

Figure 5: Complete `Geometry` data structure, showing optional point-related data.

2.3 Point Storage

Before completing the design for the individual `Primitive` data structures, an important design decision was where we should store geometric point coordinates, for those primitive types that require coordinates to define their geometry. Although we could have stored points as part of each individual `Primitive` we wanted to allow points to be shared between primitives so end-users could "stitch-together" disjoint primitive types. To support this, the top-level `Geometry` data structure includes optional "points" and "point_selection" arrays that store the 3D coordinates and selection state respectively of all the points for the entire data structure. `Primitive` objects that require point-based geometry reference the points using arrays of indices into the "points" array, so that modifying the coordinates of a point that is shared between two primitives implicitly affects each. Figure 5 demonstrates the updated `Geometry` data structure with point-related storage arrays[1]. Note that this approach simplifies some operations while complicating others — for example, algorithms that deform geometry using simple transformations can modify the points in the top-level `Geometry` object without having to know anything about the `Primitive` objects that use those points. On the other hand, operations that that add or remove points to the parent `Geometry` object must update the index arrays in individual `Primitive` instances so they continue to reference their original points. We will address those concerns in Section 2.5.

2.4 Primitive Structure

While the notion of a `Primitive` as a flat collection of generic arrays provided the flexibility we required, it still lacked the structure needed to perform automated operations such as constraint validation or the point index updates mentioned in Section 2.3. Here, we made the observation that relational database management systems do just this in a wide variety of contexts. In particular, many modern databases store data structures of arbitrary complexity using the relational model introduced in [Codd 1970]. Databases using the relational model store data objects using "relations" that are sets of "tuples" that share "attributes" — or in more common modern usage, "tables" containing "rows" or "records" that share "columns". In addition, relational databases also store metadata that define inter-table "relationships", such as a relationship between employee records in an employee table and employer records in an employer table. Collectively, this set of tables, columns, and relationships defined for a relational database is referred to as a "schema".

This notion of a relational database schema was a good starting-point for organizing the contents of a `Primitive` — in particular, it matched the sorts of parent-child relationships implicit in many primitive types, such as the top-down relationships in a polyhedron primitive: polyhedron→shell→face→loop→edge→vertex. Accordingly, we introduced our own `Table` data structure which stores a heterogeneous collection of strongly-typed named arrays using a map containing string array-names as keys, and the arrays themselves as values[2]. With this arrangement the name of an array is guaranteed unique, and array-lookups by name are fast. `Table` further constrains the lengths of all its member arrays to be equal at all times. With `Table` in place, a `Primitive` can be defined as a container for a collection of `Table` objects that conform to a specific `Schema`.

[1]We use floating-point storage for our selection arrays because we support "fuzzy" selections.

[2]In practice these are actually reference-counted shallow copies for efficiency.

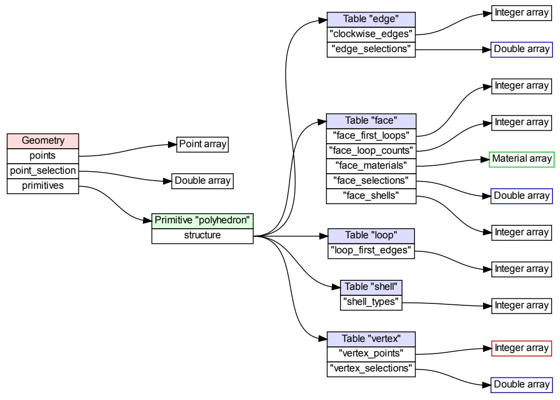

Figure 6: A `Primitive` is a collection of `Table` objects and arrays conforming to a specific `Schema`.

In Figure 6 we demonstrate a polyhedron `Primitive` that contains "type" and "structure" information:

The `Primitive` "type" is a string that uniquely identifies its `Schema`. It is used by algorithms to quickly identify the type of geometric primitive stored in a `Primitive` object. Using a string as the unique identifier makes it human-readable, easy to serialize, easy to debug, and easy to filter, as in the following pseudo-code for an algorithm that only operates on polyhedra:

```
for primitive in geometry.primitives:
    if primitive.type is "polyhedron"
        # Do polyhedron-specific stuff
```

The `Primitive` "structure" is the collection of `Table` objects that define the structure, connectivity and (optional) geometry for the primitive. The set of tables, arrays, names, and types are strictly defined by the `Schema`. As seen in Figure 6, there is a one-to-one correspondance between `Table` objects and components of a polyhedron: the "face" table contains one record for each face in the polyhedron, the "edge" table contains one record for each edge, and-so-on. The set of arrays defined for each `Table` defines the set of attributes that are available for each record. These attributes may be categorical, quantitative, or referential in nature.

2.5 Primitive Metadata

The `Schema` based `Primitive` meets all of our stated requirements for generic primitive storage, but misses-out on many possible improvements. While it is trivial to define new primitive types by defining new types of `Schema`, we wanted to go further in processing primitives. While it is easy to write algorithms that work within the confines of a known `Schema`, the set of table and array names for a known `Schema` amount to "magic numbers" that make the algorithm `Schema`-specific. Instead, we wanted to be able to write algorithms that could operate on arbitrary schemas without a priori knowledge about their specific table or array configurations.

For example, we wanted to be able to write a "Merge" algorithm that could merge `Geometry` containing schemas that the algorithm knows nothing about (Figure 7):

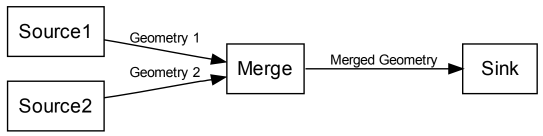

Figure 7: Merging geometric data within the visualization pipeline.

In this case, multiple `Geometry` inputs to the "Merge" data modifier can each contain arbitrary collections of `Primitive` instances, with the Merge algorithm trivially copying each instance to a single output `Geometry` data structure. While merging the `Primitive` instances is trivial, it isn't sufficient — as we saw in Section 2.3, any `Primitive` instances that contain vertices simply reference the shared vertices stored in the top-level `Geometry`, and it is necessary to offset those vertex references in the merged output. Thus, the Merge algorithm needs to know which `Primitive` arrays reference vertices and which don't. While we could have used a special array-naming scheme to identify these arrays, we did not want to impose arbitrary constraints on `Schema` authors' naming choices, and we did not want to prevent a `Schema` from containing more-than-one array that referenced points. Instead, we opted to use a generic metadata capability to "tag" arrays. Each array in a `Primitive` can contain an arbitrary collection of "metadata", composed of string name-value pairs. Using metadata, we can "tag" special attributes of individual arrays. For our Merge use-case, we defined a "domain" tag that can be used to specify that "the domain of array X is indices into array Y". Every array that references the `Geometry` points array is tagged with the domain tag, allowing the Merge algorithm to search for all of the point-index arrays in a `Primitive`, offsetting the indices in those arrays as appropriate, and without any knowledge of any specific primitive schemas.

2.6 Primitive Attributes

While a `Schema` provides everything needed to define the structure and connectivity of a primitive type, there are many other optional attributes typically assigned to geometric primitives, including per-vertex colors, normal vectors, and texture coordinates to name just a few. While we could have defined these attributes formally as part of the `Schema`, we wanted to provide researchers, developers and end-users with the flexibility to define as many or as few of their own attributes as they might like.

To accomplish this, we provide a second collection of `Table` instances in each `Primitive` that contains user-defined attribute arrays. The set of attribute `Table` instances is defined by the `Schema`, but the collection of arrays in each `Table` is user-defined. Because every array in a `Table` is constrained by-definition to be the same length, attributes are always defined consistently across a primitive.

In Figure 8, we see the polyhedron `Schema` in its full generality, with a user-defined per-face "texture_name" array, and per-vertex "color" and "texture_coordinates" arrays. Note that while we could have stored user-defined attributes in the same `Table` instances as the rest of the `Primitive`, we opted not to for two reasons: first, we did not want user-defined attribute arrays to "compete" with the `Schema` for array-names. In particular, we did not want `Schema`-defined ar-

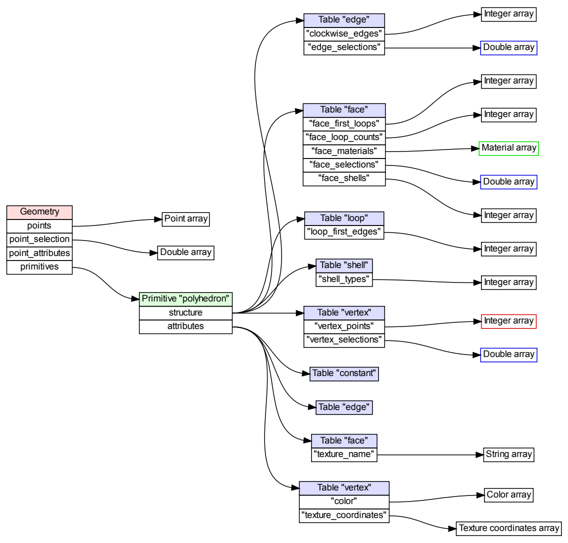

Figure 8: Complete polyhedron `Primitive` with user-defined texture, color, and texture coordinate array attributes.

rays to clash with existing user-defined arrays, if the `Schema` ever grew over time. Second, we found that it wasn't always necessary or desirable to have a one-to-one mapping between the structure and the attributes of a `Primitive` — for example, note from Figure 8, that the polyhedron `Schema` has a "constant" attribute `Table`, but no corresponding "constant" structure `Table`. This reflects the fact that, while it is useful to store user-defined per-polyhedron attributes, there is no corresponding structural requirement for per-polyhedron data. Conversely, you will also see that while the `Schema` defines a structural "loop" `Table` that stores polyhedron face loop information, we have yet to find a circumstance where we wanted to store user-defined per-loop attributes. Thus, the `Schema` currently omits support for a "loop" attribute `Table`.

3 Application Case Study

3.1 Scripting and Serialization

In the past, support for embedded scripting languages such as Python[Python 2008] meant that we had to manually write integration code to marshal each C++ primitive type across language boundaries. With the universal data structures, this code only had to be written once for the `Geometry`, `Primitive`, and `Table` classes, and now supports all primitive types for the foreseeable future. Similarly, our original serialization code had to be updated for each new primitive type added; this was extremely error-prone, since it was easy to inadvertently overlook saving or loading parts of the data. Now a single generic operation handles serialization for all primitive types — we simply walk the tree of `Geometry`, `Primitive`, `Table`, and array objects, serializing each in-turn. Loading serialized geometry is similarly easy, since it is a simple matter of instantiating the same objects in reverse order from the serialized data.

3.2 Validation and Creation

Although newly-added primitive types are automatically serialized without any code changes, care must be taken for versioning and validation — because the contents of a newly-loaded `Geometry` are based entirely on the `Schema` in effect at the time the geometry was saved, additions or modifications to the `Schema` can cause potential problems. To address problems of versioning, we need robust validation for `Primitive` objects - that is, we should not assume based on its schema type-string that a `Primitive` will contain a particular configuration of tables or arrays, since individual tables and arrays may be added, renamed, or removed as a schema evolves. To address this, we currently provide a set of validation functions for a set of common primitive types that are defined for the application. These validation functions check to confirm whether a `Primitive` instance contains a complete, correct array configuration for a specific schema. The process confirms that the `Primitive` contains every `Table` defined by the `Schema`, that structure and attribute `Table` objects with matching names are the same length, that each `Table` contains the arrays required by the `Schema`, that each array stores values of the correct type and contains the expected metadata, and that the length of each array is consistent with the rest of the data. Further validation confirms that array indices are not out-of-bounds, and that the `Geometry` contains point data if any of its primitives contain point indices. We recognize that this approach is only a partial solution to the validation problem, since it means that validation is only defined for a set of "builtin" primitive types that are defined at compile time. In future work we plan to make primitive validation a type of plugin, so that validation will be available for new primitive types that are deployed via plugin.

Similarly, there are many places in the code where data sources create instances of a specific primitive type. Rather than duplicate the logic to create a specific schema and all its tables and arrays at multiple points in the code, we provide schema-specific creation functions that centralize this logic.

3.3 Sample Schemas

Currently, we have defined schemas, creation, and validation functions for all of the primitives defined by RenderMan (bicubic patches, bilinear patches, blobby implicit surfaces, cones, cubic curves, cylinders, disks, hyperboloids, linear curves, NURBS patches, paraboloids, points, polyhedra, spheres, and tori), plus NURBS curves and Newell Teapots.

3.3.1 Teapots

In contrast to the polyhedron `Schema` presented in the preceding sections, one of our simplest schemas defines storage for the classic Newell Teapot. As always, keep in mind that we are defining teapots as a distinct primitive type, rather than creating teapot-shaped surfaces using collections of patches or polygons. This allows teapot primitives to be represented extremely compactly: since the shape of the teapot is implicit, we need only supply a few parameters per teapot, such as the choice of surface material and a transformation matrix. We define the teapot schema so that a single primitive can contain an arbitrary number of teapot instances. This approach allows us to take full advantage of the benefits of an array-based encoding, so that an arbitrarily large number of teapot instances can be represented compactly with just a few arrays. We outline the teapot `Schema` in Figure 9:

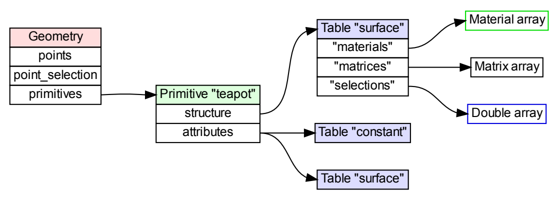

Figure 9: Teapot `Schema`

The teapot structure consists of a single `Table` containing "surface" (per-teapot) data: an array of surface material definitions, an array of 4x4 transformation matrices, and an array to keep track of selection state. Each teapot instance will have one entry in each of the "materials", "matrices", and "selections" arrays, controlling its surface appearance, transformation, and selection-state respectively. The `Schema` defines additional storage for "constant" (global) and "surface" (per-teapot) attributes. Note that the `Geometry` "points" and "point_selection" arrays are not needed, since the teapot primitive geometry is completely implicit.

We have defined several primitive schemas that follow this model of simple, "implicit" (no points) primitives, including cones, cylinders, disks, hyperboloids, paraboloids, spheres, and tori.

3.3.2 Bicubic Patches

In Figure 10 we demonstrate a more complex `Schema` that encodes bicubic patches. As with the teapots `Schema`, we store zero-to-many patches in a single `Primitive` object for efficiency:

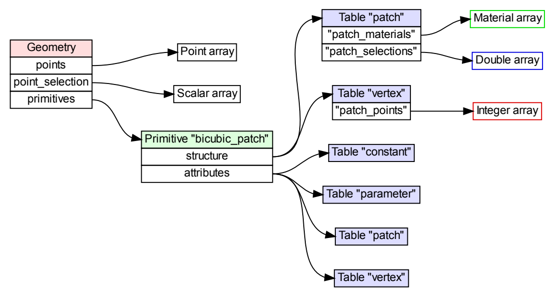

Figure 10: Bicubic patch schema

As you would expect, the "patch" `Table` will contain one row for each patch instance in the primitive. Because each patch is always composed of 16 control vertices, the number of rows in the "vertex" table will always be 16×the number of rows in the "patch" table, and no further data is required for random access into the "patch_points" array. The bicubic patch attributes include "constant" (global), "patch" (per-patch), "vertex" (per-control vertex) and "parameter" (per-parametric-corner) attributes. The "vertex" attribute `Table` must contain 16 rows per patch and the "parameter" `Table` must contain 4 rows per patch.

We have defined many other primitive schemas that store point-based geometries in similar fashion, including bilinear patches, NURBS patches, cubic curves, linear curves, and NURBS curves.

3.3.3 Implicit Surfaces

Finally, Figure 11 provides an example of one of our most complex and unusual schemas to-date, one which can store implicit surfaces compatible with RenderMan "blobbies":

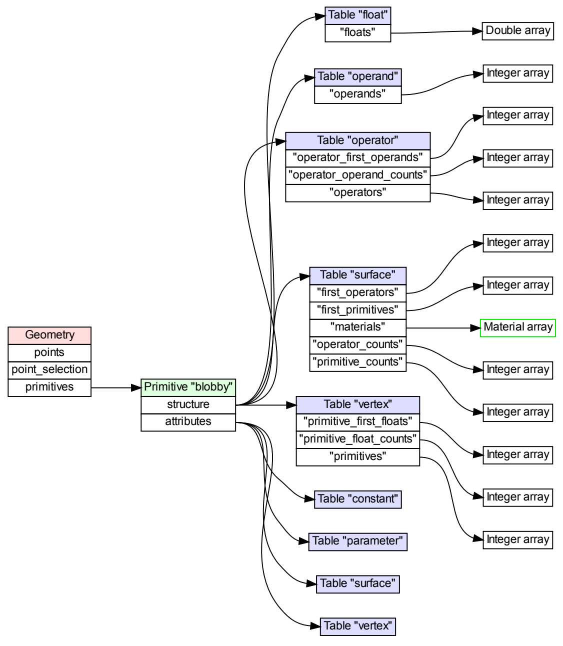

Figure 11: Implicit surface schema

In this case, the schema encodes a tree of assembly-language-like opcodes and operands that are used to compute implicit surfaces.

3.3.4 Bezier Triangles

Since adopting the new relational data structures, Ashish Myles of the Florida Surf Lab (`http://www.cise.ufl.edu/research/SurfLab`) has implemented Bezier triangle primitives using the new data structures, see Figure 12.

3.4 Runtime Schemas

Although the bulk of end-users will do their work using the "builtin" primitive schemas, the flexibility of the universal data structure provides researchers and power-users with a unique capability to create their own new primitive types at runtime. For K-3D, this means creating three types of pipeline component: "sources" that can create instances of a new primitive type; "modifiers" that perform operations on the new type, and "painters" that can render the new type using APIs such as OpenGL and RenderMan. Because K-3D provides scripted / programmable versions of each of these pipeline component types, end-users are free to introduce new primitive types "on-the-fly" at runtime using scripting languages such as Python:

- A researcher working with new or modified primitive types (point based surfaces, new subdivision surface types, new implicit surface types, etc) can introduce them into K-3D with a minimum of effort, focusing on

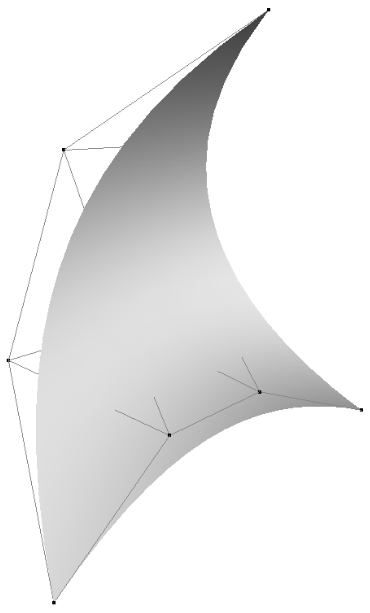
Figure 12: Bezier Triangle primitive support

the definition and rendering of their new primitive while benefitting from the existing infrastructure (the graphical user interface, automatic serialization, a large body of existing sources and modifiers, etc).

- A "power-user" can define their own special-purpose primitives, such as a "railroad track" primitive that combines a low-resource parameterization (a simple description of the position of the track) with sophisticated painters that perform "complexity expansion", creating rails and ties at render-time.
- For users working with RenderMan or other render engines, custom primitives can be defined as proxies for procedural or implementation-defined primitives, so that visually-complex rendered primitives have low-resolution counterparts in the application user interface.

In the following example, we create a new "cube" `Schema` at runtime using a Python script, and populate it with a random collection of cubes. Of course, creating a specialized primitive type for cubes is a contrived example, but it allows us to present a complete, working example in the space available. Note that, like the "teapot" `Schema` outlined in Section 3.3.1, we are not simply creating cubes using polyhedra — rather, we are defining storage for primitive cubes with implicit geometry. Each cube instance will be a unit cube centered on the origin, with just a surface material and transformation matrix to be stored per-cube. This script would be used with a pipeline programmable source component, which executes a Python script to create a `Geometry` instance as output. In the example we define "surface" (per-cube) attributes, and add a sample "Cs" color attribute that can be used to assign per-cube colors:

```
import k3d
import random

# Construct a custom "cube" schema:
cubes = Output.primitives().create("cube")
matrices = cubes.structure().create("matrices",
  "k3d::matrix4")
materials = cubes.structure().create("materials",
  "k3d::imaterial*")
surface = cubes.attributes().create("surface")
color = surface.create("Cs", "k3d::color")

# Add a bunch of cubes at random:
for x in range(-10, 11):
  for y in range(-10, 11):
    matrices.append(
      k3d.translate3(x, y, 0)
      * k3d.scale3(random.uniform(0.2, 1)))
    materials.append(None)
    color.append(k3d.color(random.uniform(0, 1),
      random.uniform(0, 1), random.uniform(0, 1)))
```

This script handles creating the new cube primitive at the beginning of a pipeline. At the end of the pipeline, we typically need a "painter" that can render the cube primitive using OpenGL or RenderMan. As before, we have a scripted pipeline sink that executes a script to perform its work. Thus, we can create a script that uses the Python bindings for OpenGL [PyOpenGL 2008] to render the new "cubes" `Schema`:

```
import k3d
from OpenGL.GL import *

glPushAttrib(GL_ALL_ATTRIB_BITS)
glFrontFace(GL_CW)
glCullFace(GL_BACK)
glEnable(GL_CULL_FACE)
glMaterial(GL_FRONT, GL_AMBIENT, [0.1, 0.1, 0.1])
glMaterial(GL_FRONT, GL_SPECULAR, [0, 0, 0])
glMaterial(GL_FRONT, GL_EMISSION, [0, 0, 0])

for primitive in Geometry.primitives():
  if primitive.type() != "cube":
    continue

  matrices = primitive.structure()["matrices"]
  surface = primitive.attributes()["surface"]
  Cs = surface["Cs"]

  for i in range(len(matrices)):

    matrix = matrices[i]
    color = Cs[i]

    glMatrixMode(GL_MODELVIEW)
    glPushMatrix()
    glMultMatrixd(matrices[i].column_major_list())

    glMaterial(GL_FRONT, GL_DIFFUSE, [color.red,
        color.green, color.blue])
    glBegin(GL_QUADS)
    # Enumerate cube vertices here.
    glEnd()

    glPopMatrix()

glPopAttrib()
```

Observe that the script iterates over the contents of a `Geometry` object, looking at each `Primitive` in-turn, filtering-out any schemas other than the "cube" `Schema`. It then extracts arrays from the schema structure by-name and uses them to render the resulting geometry. The results can be seen in Figure 13:

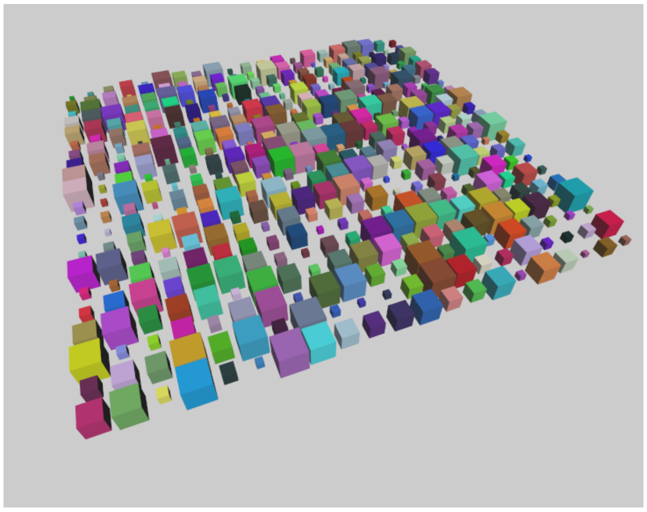

Figure 13: Scripted schema created and rendered at runtime using Python and OpenGL.

4 Conclusions & Future Work

With the new `Geometry`, `Primitive`, and `Table` data structures defined and implemented in K-3D, we have begun to reap immediate rewards. We have eliminated many detail-oriented, difficult, and error-prone maintenance tasks and decreased the size of our code-base while nearly doubling the number of geometric primitive types supported by the application.

As manufacturers continue the trend towards multi-core CPU architectures, the organization of our data structures into arrays will lend itself to efficiency, trivial partitioning, and parallel computation. We have begun to take advantage of this, using the Intel Threading Building Blocks (TBB) library [Intel 2006] to parallelize many of our embarassingly-parallel operations: for example, because geometric primitive points are all stored in a single array, it is trivial to do simple point-transformations in parallel with TBB. We will continue to develop and refine primitive schemas that work well in parallel.

In addition to CPU performance, array-based data structures make it much easier to take advantage of GPU features such as retained geometry. Many GPU APIs focus on textures (2D arrays) for data exchange with the CPU. Because we organize our primitive structure and attributes into arrays from the start, vertex coordinates, normals, texture coordinates, per-vertex colors, etc. can be passed directly to GPUs in a single call. Given the deprecation of immediate-mode rendering in OpenGL 3.0 [Segal and Akeley 2008], the importance of supporting retained-geometry APIs cannot be overestimated.

We plan to expand the use of metadata in our schemas to increase the number of operations that can be generalized to multiple primitive types. For example, an interesting challenge would be to explore whether it is possible (or practical) to add sufficient high-level connectivity metadata to create an extrusion algorithm that works with both NURBS surfaces and polyhedra, without containing any hard-coded knowledge about either.

We would like to further break-down the current distinction between primitive types that have creation and validation code built-in to the application, and primitive types that are created by third-parties. This will require pluggable validation and creation operations for new primitive types. By registering validation plugins against primitive schemas, we will be able to apply validation to new types automatically.

References

Alumbaugh, T. J., and Jiao, X. 2005. Compact array-based mesh data structures. In *Proceedings, 14th International Meshing Roundtable*, Springer-Verlag, 485–504.

Bloomenthal, J., and Wyvill, B. 1990. Interactive techniques for implicit modeling. In *SI3D '90: Proceedings of the 1990 symposium on Interactive 3D graphics*, ACM, New York, NY, USA, 109–116.

Codd, E. F. 1970. A relational model of data for large shared data banks. *Communications of the ACM 13*, 6, 377 – 387.

Intel, 2006. Threading building blocks. http://www.threadingbuildingblocks.org.

Kobbelt, L., and Botsch, M. 2004. A survey of point-based techniques in computer graphics. *Computers & Graphics 28*, 6, 801–814.

Pixar, 2005. The renderman interface version 3.2.1, November.

PyOpenGL, 2008. Pyopengl home page. http://pyopengl.sourceforge.net.

Python, 2008. Python home page. http://python.org/.

Schroeder, W., Martin, K. M., and Lorensen, W. E. 2006. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 4th ed. Kitware, Inc.

Segal, M., and Akeley, K. 2008. The opengl graphics system: A specification (version 3.0 - august 11, 2008).

Shead, T. M., 2005. K-3d home page. http://www.k-3d.org.

Shead, T. M., 2008. K-3d visualization pipeline. http://www.k-3d.org/wiki/Visualization_Pipeline.

Zorin, D. 2006. Modeling with multiresolution subdivision surfaces. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, ACM, New York, NY, USA, 30–50.