VSXu, Abstract

Jonatan Wallmander
Vovoid Media Technologies,

Abstract—**A visual programming language for real time (OpenGL) graphics**

I. Introduction

For a programmer and designer, creating real-time interactive graphics with resulting high performance is a complex and tedious task often taking so long that it stifles creativity. There are many solutions to the problem aimed primarily at artists. However many of these are game engines with fixed feature sets, and a substantial portion are proprietary and or single-platform. For programmers and advanced digital artists (wanting low-level control) there are few options. Programmers writing code can choose from a multitude of libraries speeding up development considerably but there is still a large gap to be filled.

II. Problems With Traditional Programming Methods

A. Programmer Workflow

Programming / compiling / deploying to device / testing is very time intensive. Usually "OpenGL programming" is associated with traditional programming languages - C/C++ for instance. One writes, links and runs one's program and inspects the output visually - either it looks right or it doesn't. Thus there are few automated unit tests. This process is repeated until the desired output is obtained. This is quite laborous and tedious. Even more so on mobile devices as no emulator reflects the feature set or performance of the actual device.

B. Team workflow

Designers and project managers have limited opportunity to affect projects, since most things are created at code level.

Designers depend on developers to make alterations. Changes cannot be made in real time thus requiring long waiting periods (see problem A).

Another issue that automatically arises is the re-use of code in a sensible way. (See figure 1)

A common solution implemented in many of the aforementioned game engines is to use a scene graph and traverse it in a separate render pass. However for the case of close-to-interactive programming, a scene graph makes it difficult to inject code and generate assets in real time. Such operations must have access to the whole graph to compute relative position, lighting hierarchy etc. which adds complexity both for the programmer and the computer. A scene graph's strengths rather lie in displaying large and complex scenes where not all geometry is visible at the same time.

```
glBegin(GL_QUADS);
  glTexCoord2f(0.0f,0.0f);
  glVertex3f(-1.0f, -1.0f, 0.0f);
  glTexCoord2f(0.0f,1.0f);
  glVertex3f(-1.0f,  1.0f, 0.0f);
  glTexCoord2f(1.0f,1.0f);
  glVertex3f( 1.0f,  1.0f, 0.0f);
  glTexCoord2f(1.0f,0.0f);
  glVertex3f( 1.0f, -1.0f, 0.0f);
glEnd();
```

Figure 1. Typical OpenGL API code defining vertices and texture coordinates

III. Our Solution - VSXu

VSXu was designed as a Visual Programming Language to solve these issues while still allowing for the traditional programming method to extend it. The initial motivation was to make our OpenGL and graphics related code reusable. Code blocks are totally isolated from each other forcing the programmer to write code adhering to the architecture. As a bonus we get good cohesion and very little overhead.

The time spent for the programmer/designer, (for things VSXu can do) is roughly estimated to be at least as low as 10% compared to writing the code in C++.

Another design goal was to make the implementation OS independent. It was also made Free Software (GPLv2) to invite others to improve on the concept, use it, and most importantly: extending it by creating modules (using C/C++). A GUI was developed - (also rendered using OpenGL) to fit the engine design and also to be inspiring for the artist using it.

IV. Quick Overview of the Architecture

The system architecture is divided into 3 layers:

- modules (code-level building blocks, defining input/output parameters)
- engine (creating/destroying modules and connecting/mediating parameter values between modules)
- editor/player ("VSXu Artiste", "VSXu Player")

The engine is a standalone software library written in C/C++ which allows for easy embedding. There is an abstraction layer for effortless linking with existing code.

When running the engine, the caller can control it via a set of challenge/response human-readable text commands allowing for a very thin interface. For instance: over TCP/IP, compatible with telnet.

There are commands for creating module instances, setting modules' parameter values, connecting inputs/outputs from 2 modules together, getting statistics from the engine etc. Albeit tedious, it is theoretically possible to enter these by hand to build up a VSXu program (called a "state"). Another possibility (for the sake of argument) is to write programs in scripting languages that generate the commands for a state (PERL/PHP/Python etc.).

Even though the VSXu project is targeted at real time graphics, the engine itself is void of graphics API calls. All such are performed in modules which in theory makes it possible to run the engine on headless systems using only the command interface.

A VSXu module is external to the engine, living as a Shared Object (DLL). Each module extends an abstract class making it easy for 3rd parties to create without having to work with the engine internals.

Since the project can be considered a development platform in itself, a programmer can thus use the traditional libraries mentioned in the introduction. There are endless possibilities what can be wrapped in modules. Encapsulating a complex structure like a scene graph into a module for instance is possible however it could be argued that it's hard to manipulate using the relatively basic data types present in the engine.

A. A brief selection of existing modules (out of the over 200 currently implemented):

- Binary arithmetic
- Sound input analyzer (FFT)
- Float value oscillators (including custom-shaped curves)
- Basic Linear Algebra arithmetic
- Unit Quaternion arithmetic including SLERP
- Particle system generators, modifiers and renderers
- Mesh generators, modifiers, deformers
- OpenGL manipulators

B. Some of the data types supported:

- float/int/float array
- string
- mesh (defining 3d triangles/normals/uv etc.)
- texture
- particle systems

The engine implements parameter interpolation where possible (mainly for floats/float tuples) supporting the GUI's smoothness look and feel.

The commands required to re-create a project can be dumped from the engine and saved either as a plain text file, or as a compressed library containing all external assets such as mesh files, texture image files, sound data etc. (the latter has to be done by the engine).

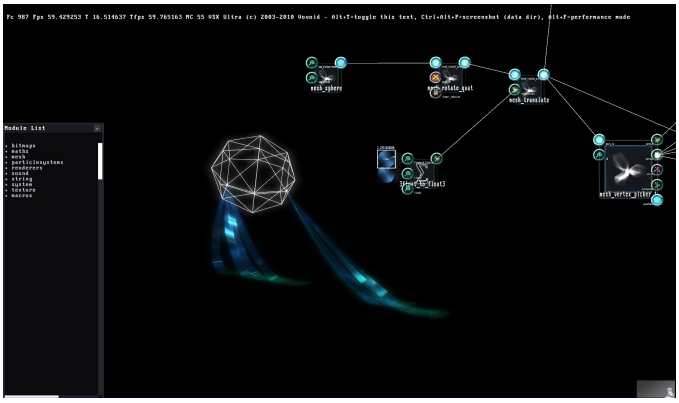

Figure 2. Tuning a cloth simulation module in VSXu Artiste "performance mode" in which the GUI is overlaid on top of the engine output

V. The GUI - VSXu Artiste

VSXu Artiste provides a zoomable UI where one can group modules together as "macros". When creating new modules the library can be searched either via a hyperbolic tree structure or by incremental text search. Modules are then created by dragging and dropping. Connections between modules are also created by drag & drop.

Each module's parameters (if not connected to another module) can be controlled with various graphical controllers including knobs, sliders, color choosers and more.

Groups of modules (and macros) can be cloned for quick programming.

Since this is also rendered with OpenGL, full-window rendering of the GUI overlaid on the output of the engine is possible (See Fig. 2).

The engine and GUI also includes animation tools with various types of interpolation. Linear interpolation, sine interpolation and 4-polynomial bezier splines.