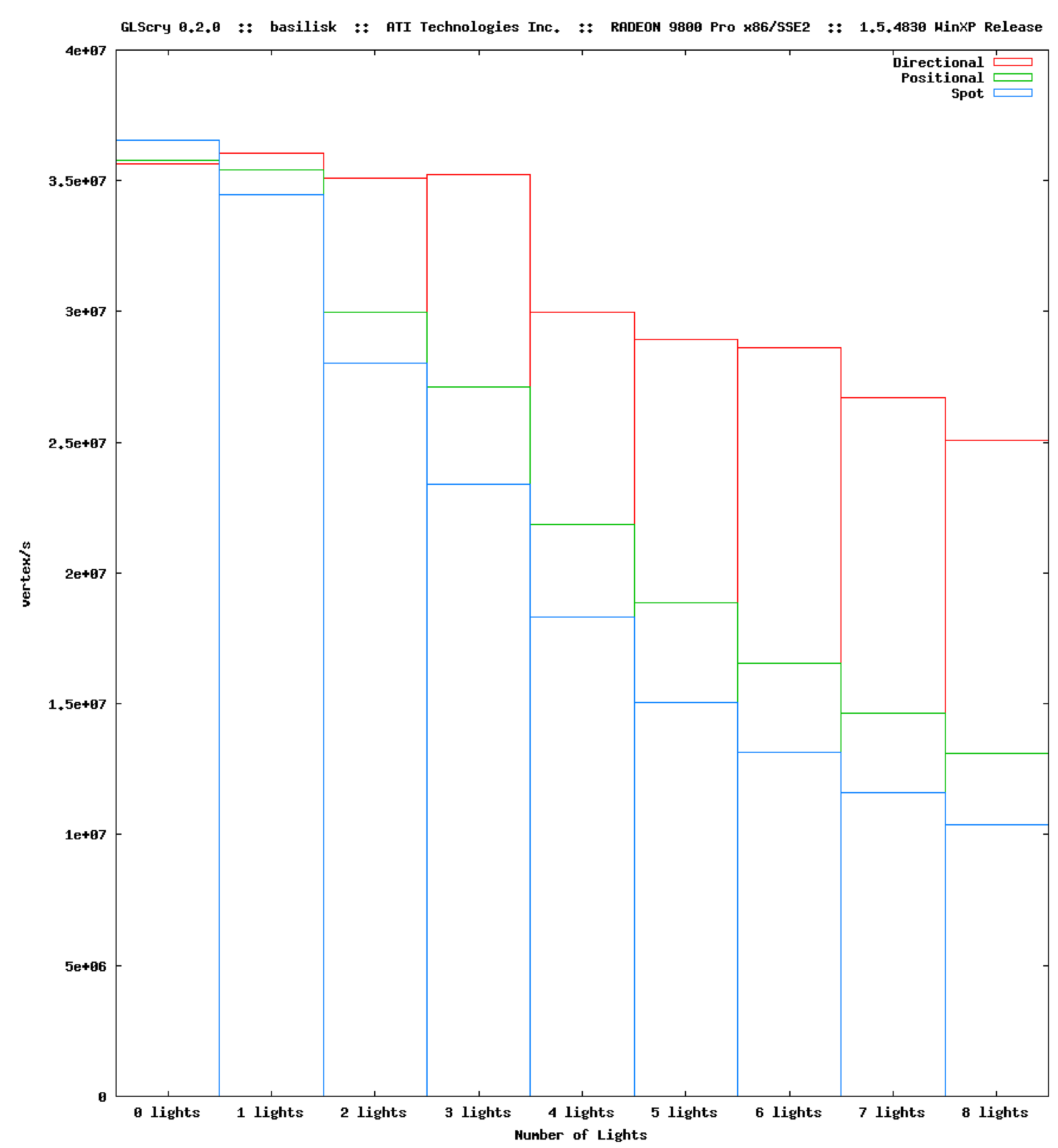
GLScry: OpenGL Performance Analysis Toolkit

Motivation
- GPU manufacturers are secretive about their specific performance characteristics.
- Existing OpenGL performance analysis tools are either specific to one subsystem or too old.
- We want one extensible framework that includes a variety of tests for features in modern GPUs including vertex cache size, existence of hierarchical Z, and cost of switching shader state versus textures.
- Automatic optimization of a scene graph based on the current display hardware.
- It should be possible to write and extend tests without having to recompile anything.

Implementation
- Python test scripts set up geometry, render states and drive Boost.Python-exported C++ measurement code.
- The native measurement code runs some OpenGL commands in a loop for some amount of time and returns a set of results (vertex rate, primitive rate, fill rate, batch rate).
- The script then graphs the results.

```
from glscry import *

geo = buildGeometry((GL_TRIANGLES, 1024),
        v=defineArray(Array_f, 2,
                      [(5, 5), (5, 6), (6, 6)]),
        n=defineArray(Array_f, 3,
                      [(1, 0, 0), (0, 1, 0), (0, 0, 1)]))

def dirLight(light):
    light.ambient = Vec4f(1, 1, 1, 1)
    light.diffuse = Vec4f(1, 1, 1, 1)
    light.specular = Vec4f(1, 1, 1, 1)
    light.position = Vec4f(0, 0, 1, 0)

def posLight(light):
    light.ambient = Vec4f(1, 1, 1, 1)
    light.diffuse = Vec4f(1, 1, 1, 1)
    light.specular = Vec4f(1, 1, 1, 1)
    light.position = Vec4f(1, 2, 3, 1)

def spotLight(light):
    light.ambient = Vec4f(1, 1, 1, 1)
    light.diffuse = Vec4f(1, 1, 1, 1)
    light.specular = Vec4f(1, 1, 1, 1)
    light.position = Vec4f(1, 2, 1, 1)
    #light.spotExponent ?
    light.spotCutoff = 45

def makeTest(buildLight, i):
    state = LightState()
    state.enableLighting = True
    for j in range(len(state.lights)):
        light = state.lights[j]
        light.enable  = j < i
        buildLight(light)

    test = VertexArrayTest('%s lights' % i, geo)
    test.addStateSet(StateSet(state))
    return test

def run(shortname, testList, type):
    line = runTests(type, testList, 10)
    generateGraph('lights_' + shortname, line, 'VertexRate',
                  xlabel='Number of Lights')
    return line

lights = range(len(LightState().lights) + 1)
lines = [
    run('dir',  [makeTest(dirLight,  i) for i in lights], 'Directional'),
    run('pos',  [makeTest(posLight,  i) for i in lights], 'Positional'),
    run('spot', [makeTest(spotLight, i) for i in lights], 'Spot') ]
generateGraph("lights", lines, 'VertexRate', xlabel='Number of Lights')
```



Renaissance: Next Generation Shading Language for GPUs

Brief History of Real-Time Shading Languages

0th generation "languages":
- Not a general-purpose language
- Use textures and special blending operations to implement some shading algorithms

Examples: special texture blend modes, register combiners

1st generation languages:
- Assembly language for register machine
- Native data type is floating point 4-vector

Examples: ARBvp, ARBfp, D3D low-level shading language

2nd generation languages:
- High-level, C-like
- Still not as expressive as we're used to on CPUs
- Often compiled into assembly language

Examples: HLSL, Cg, GLSL

Meta-programming languages:
- Use host language to express operations
- Operations on custom data types secretly compile into lower-level language
- Well-integrated facilities for passing data into shader
- Can use host language features, especially for specialization
- Require compilation in host compiler, *cannot* treat these shaders as assets

Examples: Sh, Vertigo

Motivation

When the limitations of the assembly languages became clear, the transition to a C-like language was natural. Now we're hitting the limits of the C-like languages. Half-Life 2, for example generates over a thousand shaders with a preprocess step that combines multiple independent effects.

Goals

- Introduce functional programming language concepts of higher-order functions, lambdas, automatic type inference, and referential transparency.
- Allow staged computation: generate specialized (and efficient) shaders by specifying constant values and allowing the shader to be partially evaluated in that context.
- Use human interaction design techniques to guide language specification, drawing influence from Haskell and Python.
- Hide (or at least blur) the distinction between vertex and fragment processors.

Chad Austin

Dr. Dirk Reiners

human computer interaction | HCI