Sehr geehrter Herr Rumpe,

schon seit längerem arbeitet Ihr Lehrstuhl an Monticore, einem Tool für die Entwicklung von *embedded DSLs*. Ich habe schon seit längerem ein Konzept für eine solche DSL. Meine Idee wäre es diese DSL mit Ihrer Kooperation mit Monticore als Masterarbeit an Ihrem Lehrstuhl zu entwickeln.

Motivation:

Es geht darum, die Entwicklung von Visualisierungen und allgemeinen Berechnungen zu beschleunigen. Meiner Erfahrung nach geht immer dann wenn ein komplett neuer Renderer für ein beliebiges Problem geschrieben werden muss, sehr viel Entwicklungszeit darin verloren, die relevanten Daten von der GPU aus nutzbar zu machen. Mit relevanten Daten meine ich hier Daten, die in Datenstrukturen des Hauptprogramms im Arbeitsspeicher liegen. Das Symptom welches daraus resultiert ist, dass oft allgemeine Renderer geschrieben werden, die alles können wollen, und beliebig konfigurierbar sind, anstelle von einfachen Renderern, die nur das machen wofür sie geschrieben wurden. Auswirkungen hat dies dann in der Performance und Komplexität, wenn das gewünschte Ergebnis nur wenig Anspruch auf Photorealität hat.

Einfach um mal ein Beispiel zu nennen, bei dem ich stark vermute dass dieser Zusammenhang hier existiert. Das Spiel Broforce hat triviale Grafik, allerdings keine trivialen Ansprüche an die Grafikleistung. Mit der DSL die ich hier vorstelle, wäre es wesentlich einfacher als mit üblichen Techniken einen sehr effizienten Renderer für die Visualisierung der Spiel-Daten zu entwickeln.

Zum besseren Verständnis der DSL erläutere ich kurz vereinfacht, wie die Grafik-Rendering-Pipeline auf modernen GPUs funktioniert.

Zu Beginn benötigt die Grafikkarte eine Punktwolke aus Daten, welche die Vertices der 3D Geometrie (Mesh) repräsentieren. Jeweils drei Vertices zusammen ergeben ein Dreieck auf der Oberfläche des Mesh. Vertices können neben ihren Koordinates beliebige zusätzliche Daten speichern. Die Kodierung und das Layout dieser *Attribute* im Speicher wird vom Programmierer festgelegt.

Als nächstes werden die Vertices im Vertexshader verarbeitet. Der Vertexshader ist hier ein Rechenkernel, der für jeden Vertex ausgeführt wird. Aufgabe des Vertexshaders ist es, die Vertexpositionen entsprechend der Kameraposition und der der perspektivischen Verzerrung zu transformieren. Bei OpenGL müssen die Positionen in den Bereich [-1; 1] transformiert werden, um sichtbar zu sein. Die Eingabedaten bleiben hier unverändert im Speicher. Der Vertexshader hat reinen Lesezugriff. Der Vertexshader bestimmt auch, wie die anderen Attribute, wie zum Beispiel Texturkoordinaten, verarbeitet werden.

Die *Rasterization-Stage* ist ein nicht-programmierbarer teil der Rendering-Pipeline. Bis zu dieser Stage sind alle Daten nur pro Vertex definiert. Dreiecke

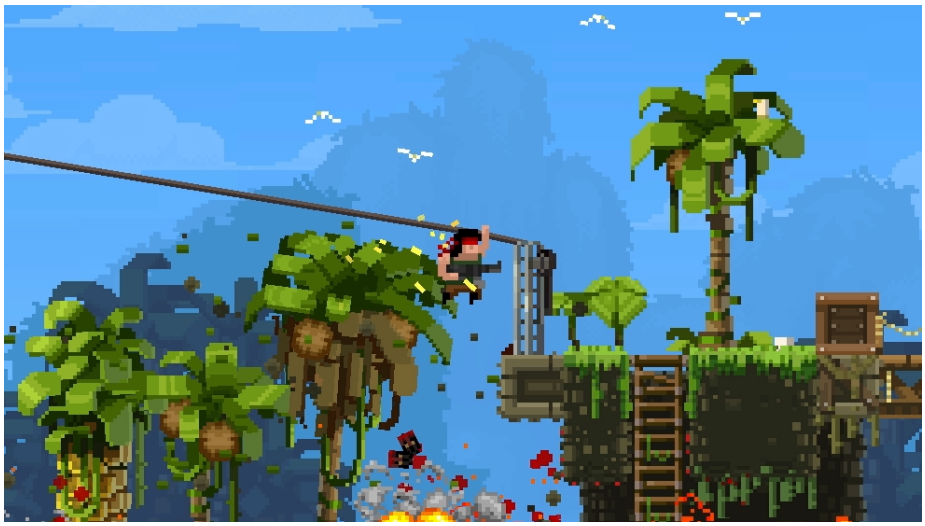

Figure 1: Simple rendering with high CPU usage

haben ihre Daten an allen Ecken definiert, und der Rasterizer hat die Aufgabe, diese Daten über das Dreieck perspektivisch korrekt zu interpolieren, so dass anschließend ein Wert pro Pixel existiert.

Der nächste Schritt in der Pipeline ist der Fragmentshader (bei DirectX auch Pixelshader genannt). Dieser berechnet aus seinen Eingaben nun letztendlich die finale Farbe für die Darstellung des Pixels. Auch dies ist ein programmierbarer Kernel mit beliebig vielen Eingaben. Der Fragmentshader übernimmt zum Beispiel die Beleuchtung und Texturierung. Besonders für die Texturierung ist die Renderingpipeline sehr hilfreich, da dieser viele Schritte für Mipmapping automatisch übernimmt. Eine einfache schleife in c++ könnte diesen Teil deshalb nicht so leicht ersetzen.

Die Kernidee meiner DSL ist es den Shadercode (GLSL), dort in den C++ code zu integrieren, wo die Shader-Eingabedaten, im lokalen Scope liegen. Der DSL Compiler soll dann die Deklarationen in GLSL generieren und die OpenGL Aufrufe in c++, die dazu Notwendig sind die Daten an das Shaderprogramm zu übergeben.

Im konkreten Beispiel sieht das wie folgt aus:

Lokale oder globale Variablen in C++ als Sicht auf die Daten:

```
ArrayBuffer<glm::vec4> positions;
ArrayBuffer<glm::vec4> colors;
glm::mat4 modelViewMat;
glm::mat4 projMat;
```

`ArrayBuffer` ist hier ein Typ vergleichbar mit `std::vector` mit zusammenhän-

gendem Speicher für alle Elemente. Der Hauptunterschied zu `std::vector` ist allerdings, dass der Speicher auf der Grafikkarte liegt. Ein ArrayBuffer speichert pro vertex Attribute. `modelViewMat` und `projMat` sind Variablen, die zwar veränderbar sind aber pro Aufruf des Shaderprogrammes nur noch den selben wert für sowohl Vertex-, als auch Fragmentshader haben, und das über alle Vertices und Pixel hinweg. Diese Daten heißen bei OpenGL `uniform`.

So könnte die Anbindung der Daten an den GLSL code aussehen:

```
// this here is normal c++ code

SHADING_DSL(R"dsl(
  uniforms {
    modelView = modelViewMat;
    proj = projMat;
  }
  attributes{
    a_vertex = vertices;
    a_color = colors;
  }
  vertexMain{
    gl_Position = proj * modelView * a_vertex;
    v_color = a_color;
  }
  vertexOut{
    out vec4 v_color;
  }
  fragmentMain{
    color = v_color;
  }
)dsl");
```

Für c++ ist die DSL einfach nur ein mehrzeiliges Stringliteral. Im Block `uniforms` und `attributes`, werden die die Namen der c++ Variables benutzt um die *uniforms* und *attributes* für den shader zu definieren. Also von dem Datentyp von `modelViewMat` (`glm::mat4`) leitet sich das Uniform `"uniform mat4 modelView"` ab. Von der Variablen `vertices` (`ArrayBuffer<glm::vec4>`) leitet sich das Attribut `"attribute vec4 a_vertex"` ab. In `vertexMain` und `fragmentMain` befindet sich GLSL code. Sowohl der code in `vertexMain` als auch in `fragmentMain` soll alle Variable die in `uniforms` definiert worden sind nutzen können. `vertexMain` soll zusätzlich noch die Variablen aus `attributes` nutzen können und in die variables aus `vertexOut` schreiben können und `fragmentMain` soll aus den Daten aus `vertexOut` lesen können.

Im Anhang ist eine Beispieldatei für ein “Hello Triangle”-Programm, welche sowohl repräsentativ den DSL-Code enthält, als auch den Code den ich für diese DSL generieren würde, damit das Programm auch korrekt und Ausführbar ist.