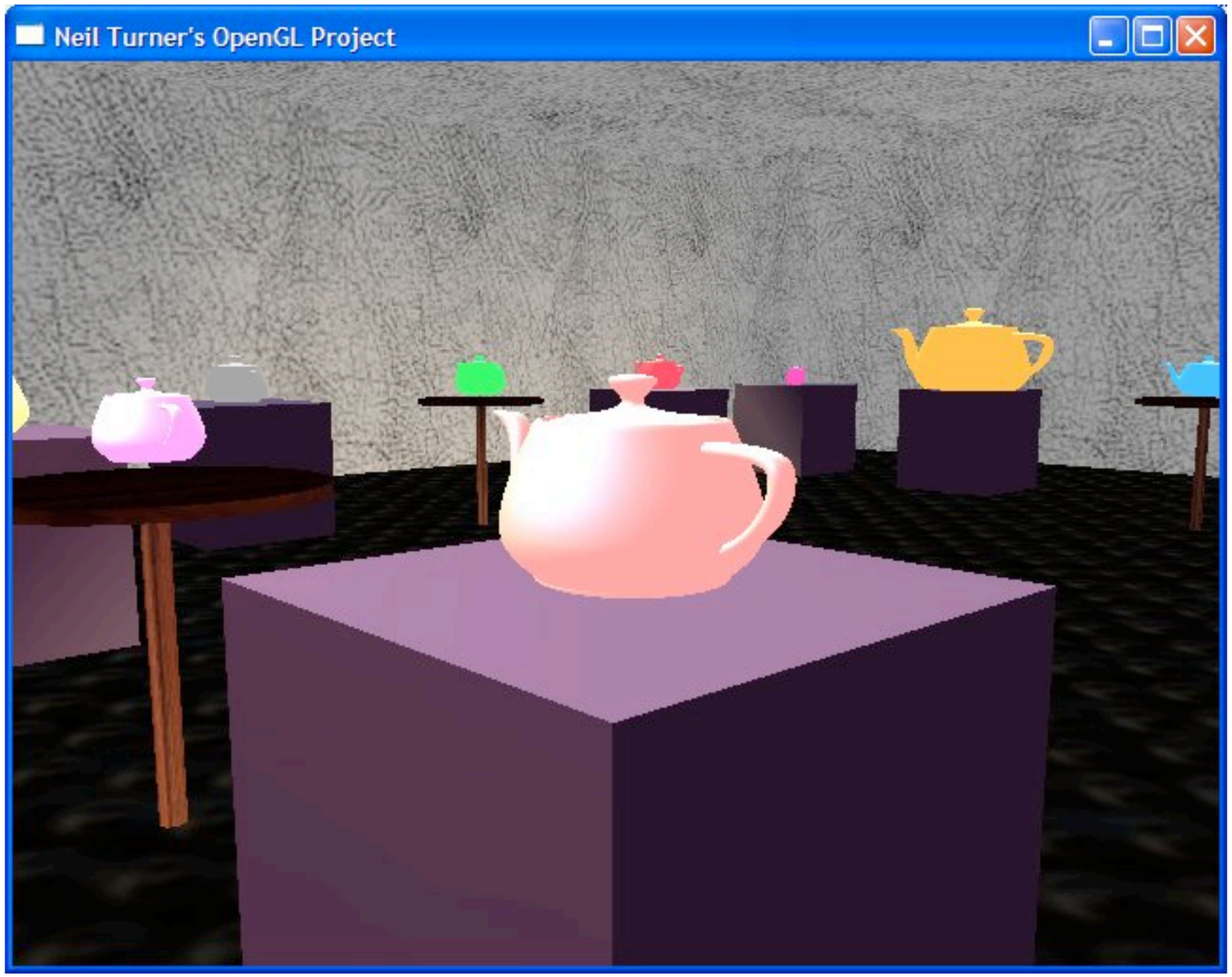
418382
Lecture 11

Pramook Khungurn

ภาพที่ OpenGL เวอร์ชันเก่าๆ สามารถสร้างได้

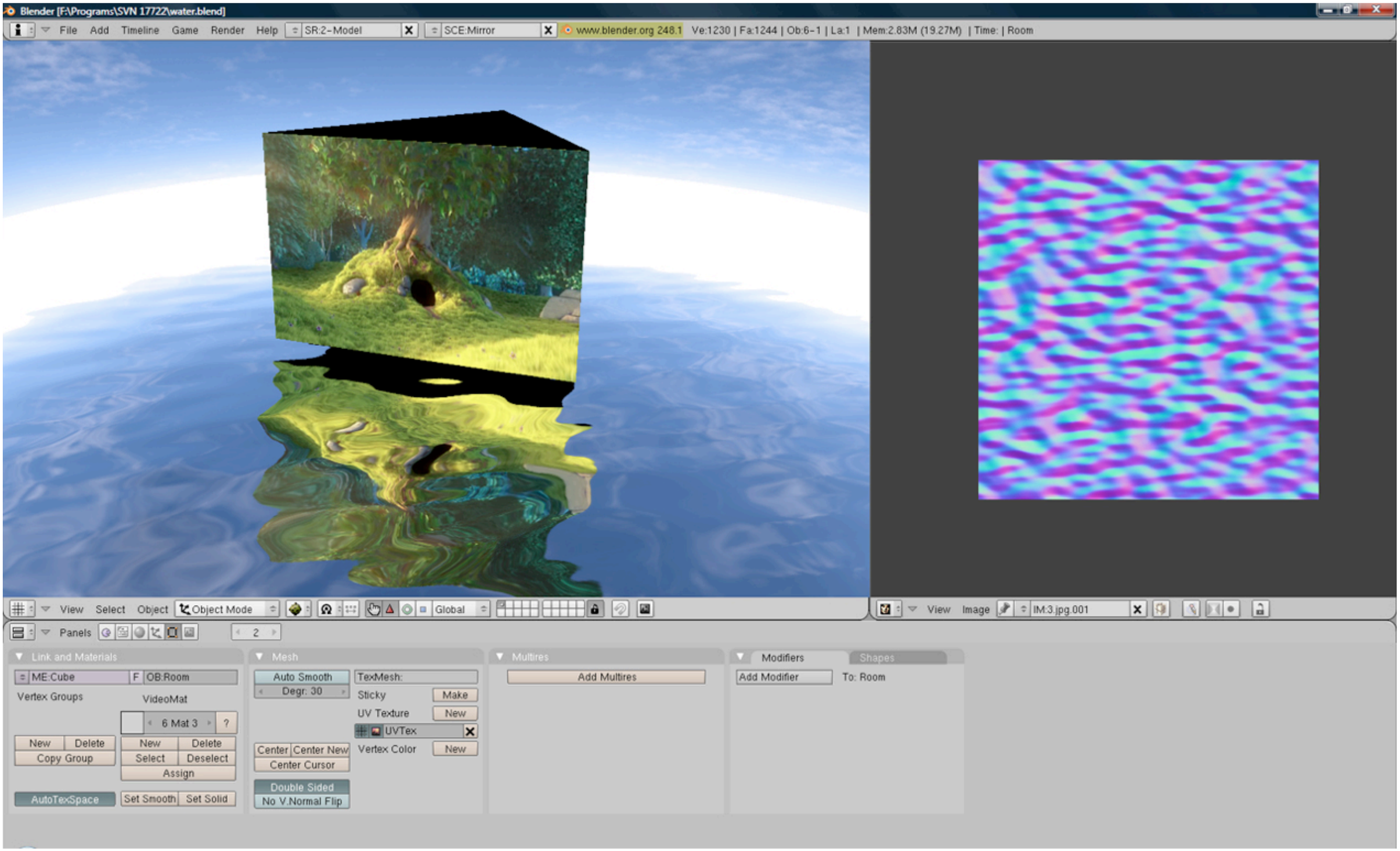


http://www.neilturner.me.uk/shots/opengl-big.jpg

ภาพที่ OpenGL เวอร์ชันเก่าๆ สามารถสร้างได้

- Shade สีของ fragment ได้แค่ตาม Phong lighting model
- ไม่มีเงา การสะท้อนแสง และ special effect อื่นๆ

ภาพที่ OpenGL ในปัจจุบันสามารสร้างได้



ภาพที่ OpenGL ในปัจจุบันสามารสร้างได้

- ไม่จำเป็นต้อง shade สีของ fragment ตาม Phong lighting model
- มี special effect ที่ซับซ้อนยิ่งขึ้น
    - เงา
    - การสะท้อนแสง
    - พื้นผิวขรุขระ
    - ฯลฯ

สาเหตุของความแตกต่าง

- แต่ก่อน เราไม่สามารถควบคุมการ์ดจอได้มากเท่าปัจจุบัน
    - การแสดงผล 3D ทุกอย่างทำด้วย hardware
- OpenGL สมัยก่อนต้องกำหนดตามความสามารถขั้นต่ำของ hardware
- แต่สมัยนี้เราสามารถโปรแกรมการ์ดจอโดยตรงได้ (แต่ไม่ทั้งหมด)

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- การ์ดรุ่นแรกๆ
  - ความสามารถทุกอย่างถูกกำหนดด้วย hardware เปลี่ยนแปลงไม่ได้
  - ผลิตโดย Silicon Graphics (SGI) และ Evans and Sutherland ราวๆ ปี 1970
  - มีความสามารถในการทำ vertex transformation และ texture mapping
  - แพงมากๆๆๆๆๆๆๆๆๆ

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- Pixel พัฒนา **Renderman Shading Language**
  สำหรับสร้างภาพยนตร์ในปลายทศวรรษ 1980

- ภาษานี้มีไว้สำหรับเขียนโค้ดสำหรับคำนวณสีของ fragment

- ทุกอย่างทำงานใน CPU
  เพราะฉะนั้นระบบจึงยืดหยุ่นและสามารถสร้างภาพคุณภาพสูงได้

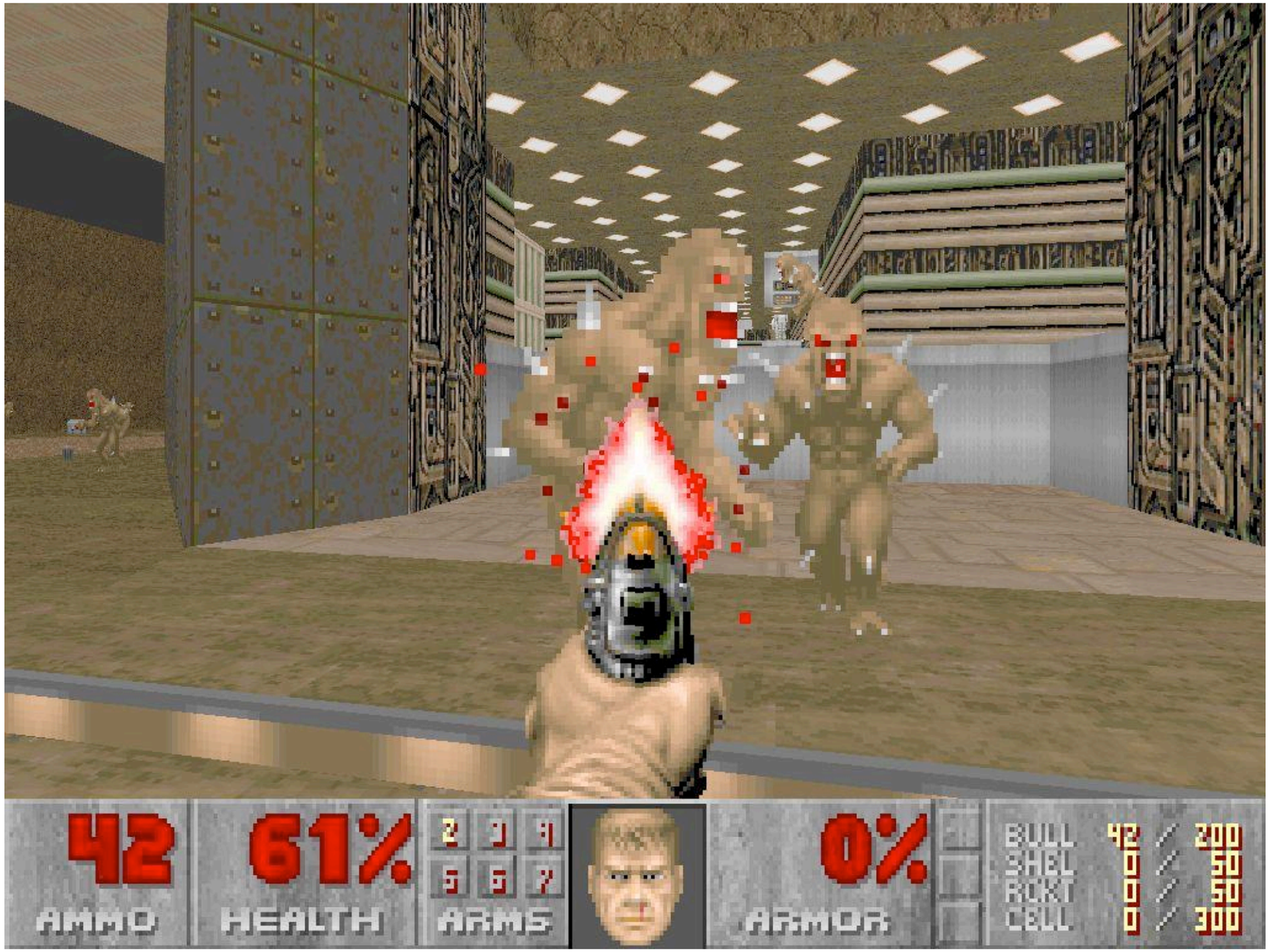
- แต่ช้า ไม่สามารถใช้กับ interactive application ได้

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- เฟรมบัฟเฟอร์
  - IBM เริ่มขายการ์ด Video Graphics Array (VGA) ในปี 1987
  - ในการ์ดมี “เฟรมบัฟเฟอร์” = หน่วยความจำที่เขียนแล้วภาพปรากฏบนหน้าจอ
  - CPU ต้องจัดการเขียนเฟรมบัฟเฟอร์เองทั้งหมด
  - ช้า ถ้าจะทำเกมต้องวาดภาพที่ไม่ซับซ้อนนัก

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์



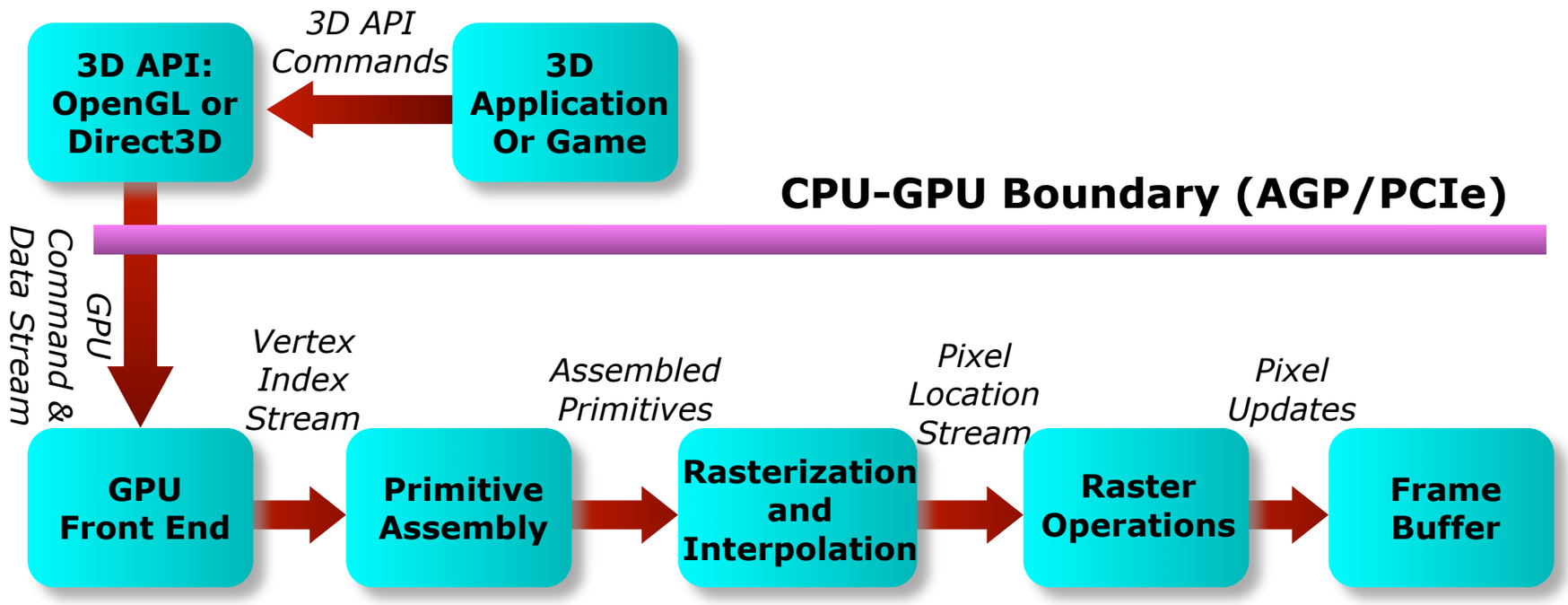
ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- GPU ยุคแรก (ถึงปี 1998)
    - TNT2 ของ Nvidia, Rage ของ ATI, และ Voodoo ของ 3dfx
    - สามารถวาดสามเหลี่ยมที่กำหนดไว้แล้ว และใช้ texture 1-2 texture ได้
        - กล่าวคือทุกอย่างต้องอยู่ใน world space(?) แล้ว
    - CPU ไม่ต้องวาดสามเหลี่ยมเองอีกต่อไป
    - ไม่สามารถ transform สามเหลี่ยมจาก object space เป็น world space ได้
    - สามารถคำนวณสีของ fragment จาก texture ได้
      แต่ความสามารถค่อนข้างจำกัด

ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- GPU ยุคที่สอง (1999 - 2000)
    - Geforce 256, Geforce2 ของ Nvidia, Radeon 7500 ของ ATI, และ Savage3D ของ S3
    - มีความสามารถในการทำ vertex transformation และ lighting ที่สมบูรณ์
    - สามารถ shade สีของ fragment ได้หลากหลายรูปแบบมากขึ้น แต่อย่างไรความสามารถก็ยังจำกัดอยู่ดี
    - สามารถ “ปรับแต่งได้” (configuration) แต่ยัง “โปรแกรมไม่ได้” (not programmable)

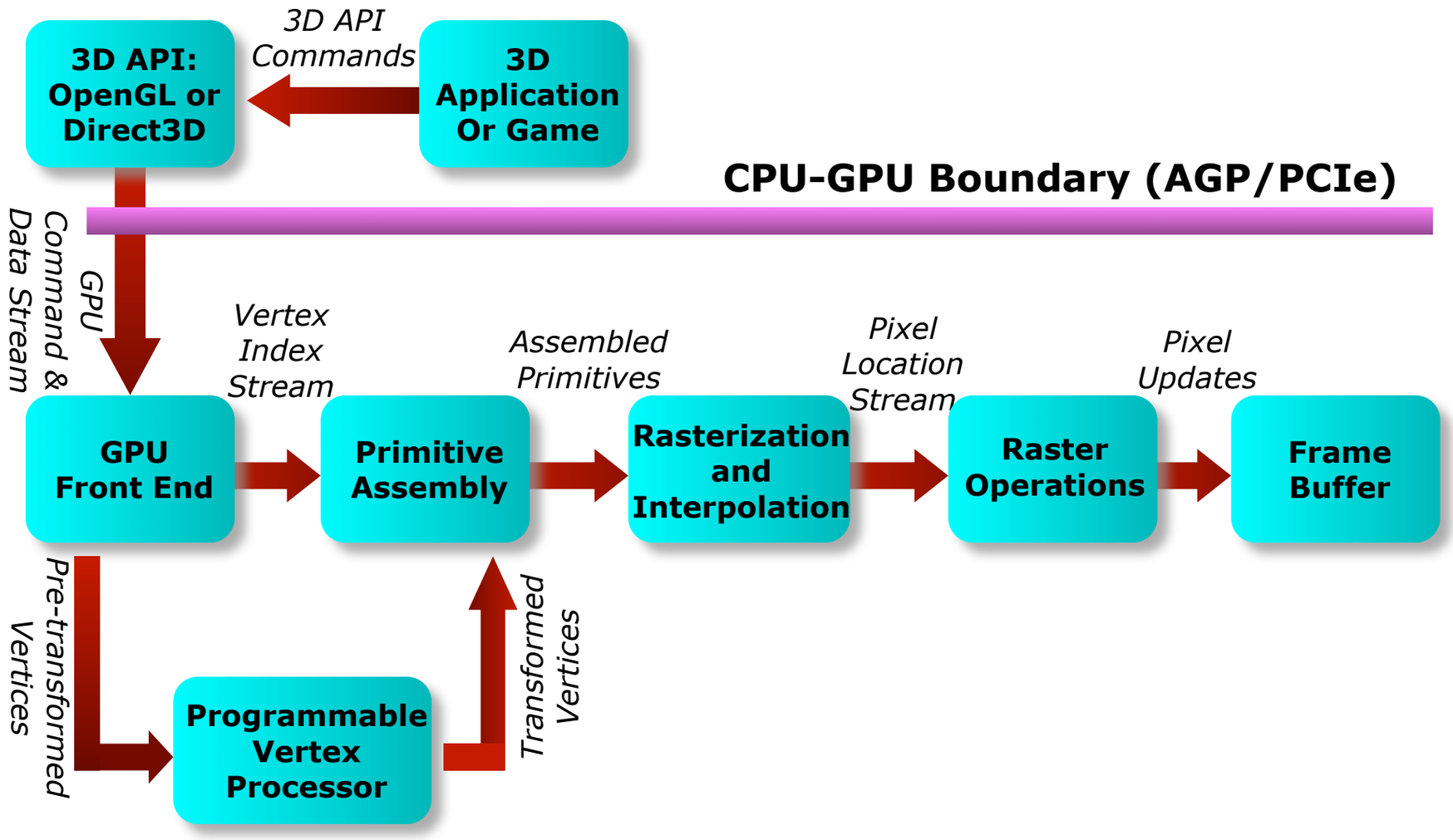
Graphics Pipeline ในยุคที่สอง



ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- GPU ยุคที่สาม (2001)
    - Geforce3, Geforce4 Ti ของ Nvidia, XBox ของ Microsoft, และ Radeon 8500 ของ ATI
        - สามารถโปรแกรมการทำ vertex transformation ได้ (vertex shader)
        - OpenGL มี extension ชื่อ ARB_vertex_program สำหรับใช้โปรแกรม vertex shader
        - สำหรับส่วนของการให้สี fragment สามารถปรับแต่งได้มากขึ้น แต่ยังไม่สามารถโปรแกรมได้จริง

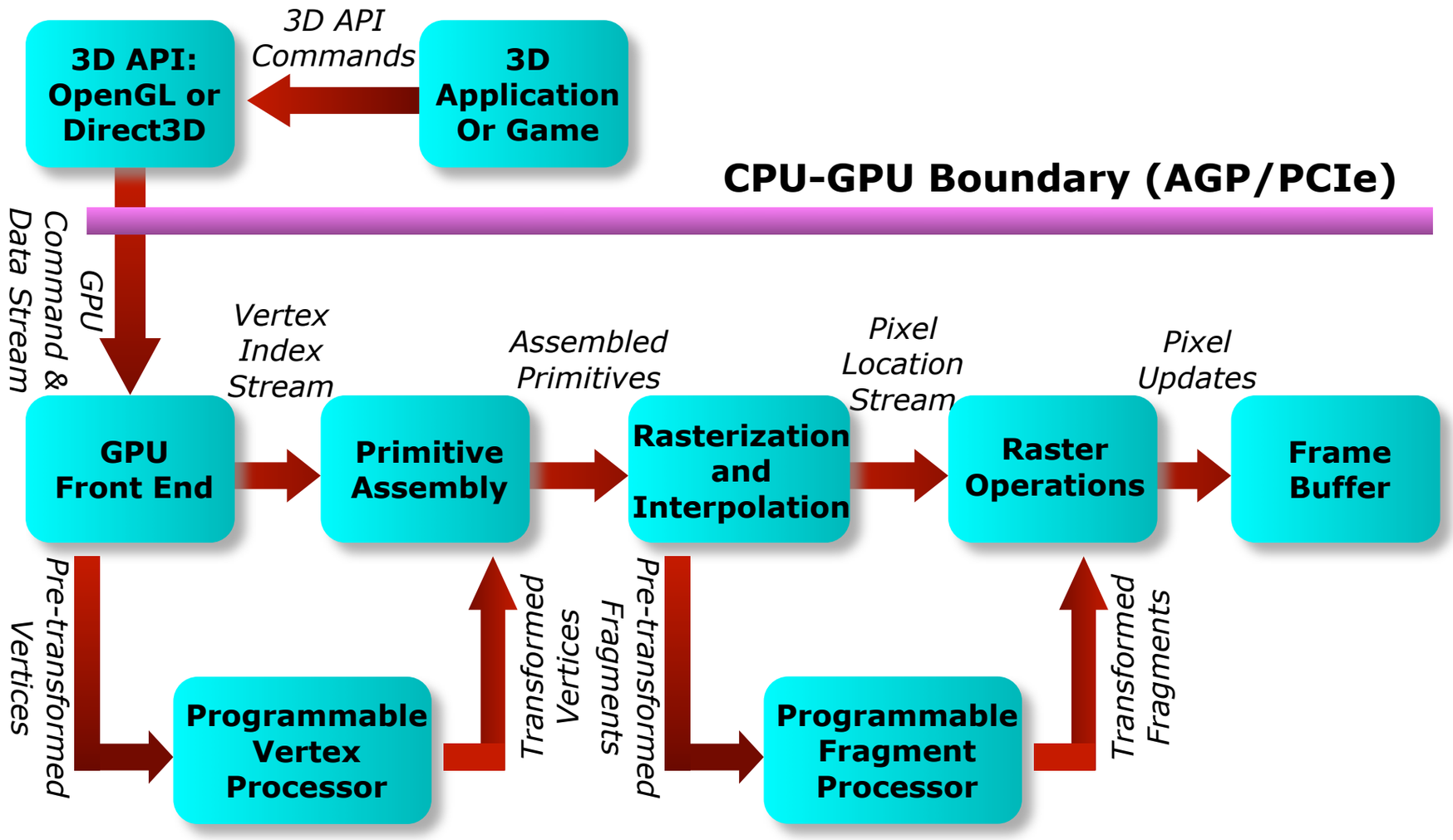
Graphics Pipeline ในยุคที่สาม



ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- GPU ยุคที่ 4 (2002 - 2007)
    - Geforce FX ถึง Geforce7 ของ Nvidia, Radeon 9700/9800 ของ ATI
    - สามารถโปรแกรมได้ทั้งการทำ vertex transformation (vertex shader) และการทำ fragment shading (fragment shader)
    - ใน OpenGL มี ARB_vertex_program และ ARB_fragment_program
    - เขียนโปรแกรม GPU ด้วยภาษา GLSL, Cg, หรือ HLSL

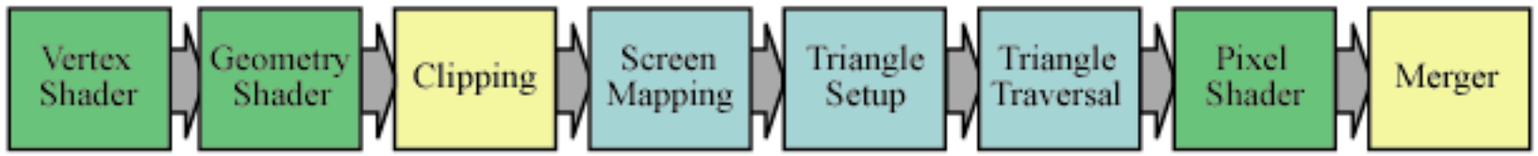
Graphics Pipeline ในยุคที่สี่



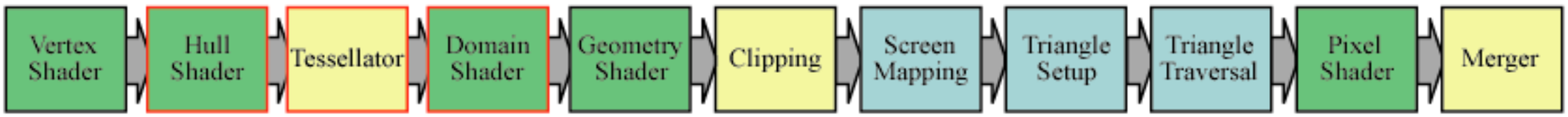
ประวัติของเทคโนโลยีคอมพิวเตอร์กราฟิกส์

- GPU ยุคที่ 5 (2007 เป็นต้นมา)
    - GeForce 8800 ของ Nvidia ขึ้นไป
    - สามารถโปรแกรม GPU เพื่อการประมวลผลอื่นที่ไม่ใช้ทางกราฟิกส์ได้
    - ในทางกราฟิกส์เองมีองค์ประกอบอื่นๆ ที่สามารถโปรแกรมได้เพิ่มเติม
        - geometry shader สำหรับสร้างรูป primitive ที่ซับซ้อนอื่นๆ
        - tesselation shader สำหรับสร้างพื้นผิวโค้ง

Graphics Pipeline ในยุคที่ห้า



หรือ



ภาพจาก http://www.realtimerendering.com/blog/direct3d-11-details-part-ii-tessellation/

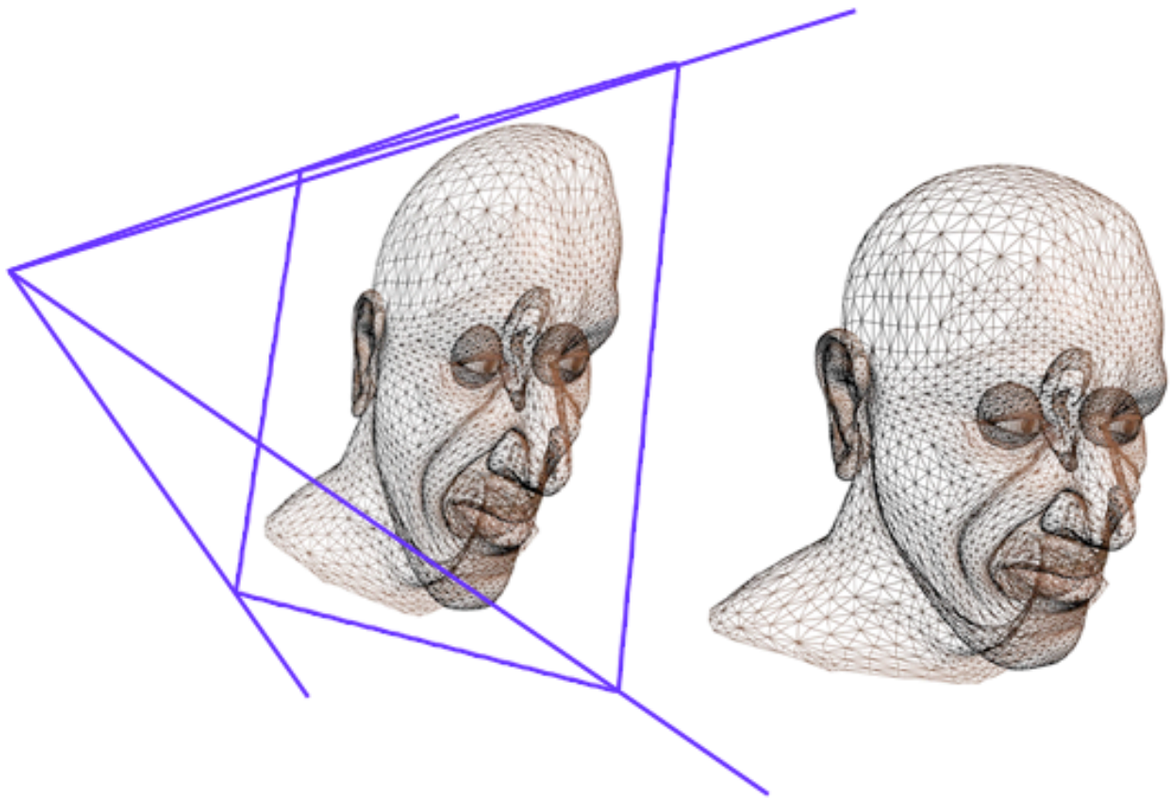
Vertex Shader และ Fragment Shader

Shader

- โปรแกรมขนาดเล็กสำหรับควบคุมส่วนประกอบส่วนหนึ่งของ GPU
  - ถ้าควบคุม vertex processor เรียกว่า vertex shader
  - ถ้าควบคุม fragment processor เรียกว่า fragment shader

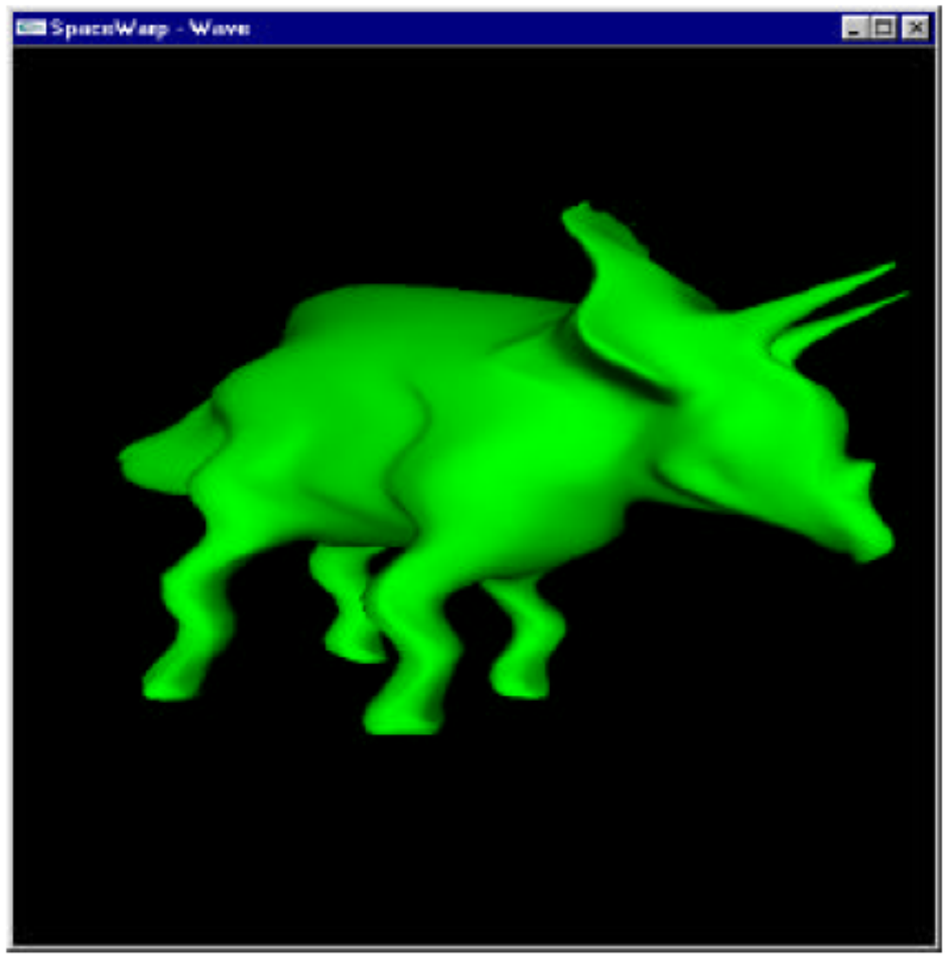
Vertex Shader

- เปลี่ยนพิกัดของ vertex ต่างๆ ให้อยู่ในระบบพิกัดของกล้อง
    - กล่าวคือจาก object space ไปเป็น clip space
- คำนวณข้อมูลอื่นๆ ที่จะถูก interpolate ตาม vertex
    - สี
    - texture coordinate
    - ฯลฯ



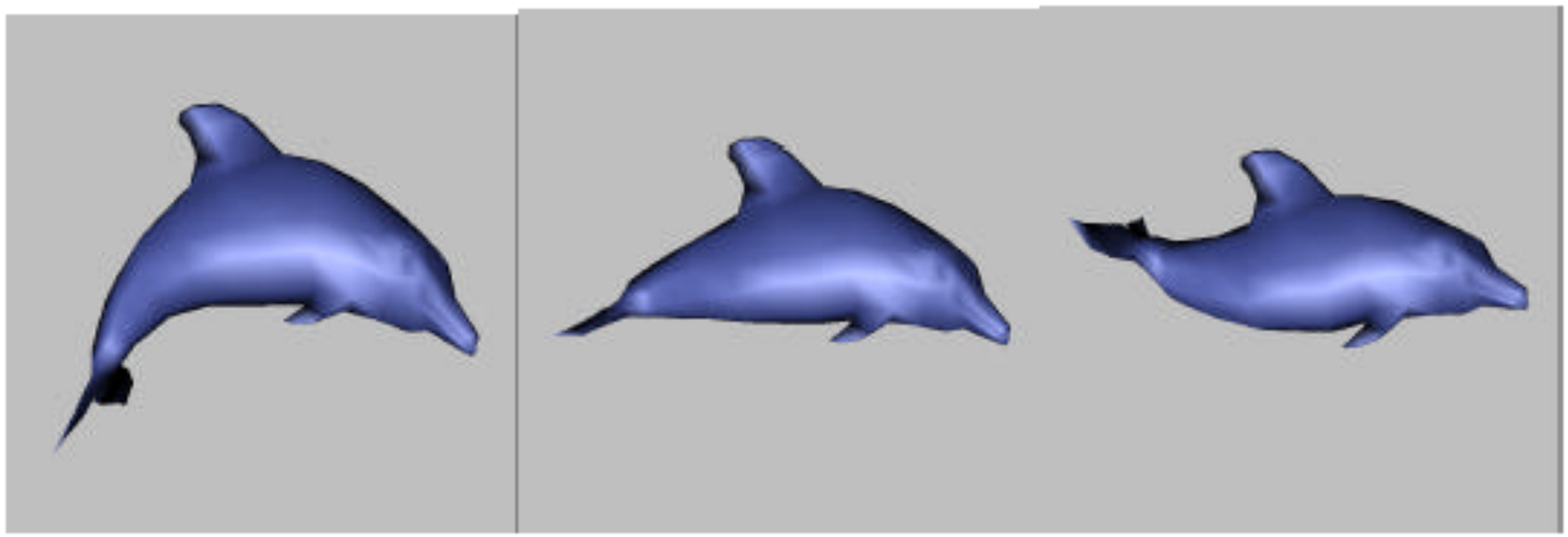
เราสามารถใช้ Vertex Shader ทำอะไรได้บ้าง?

- รูปทรงบิดเบี้ยว ธงสะบัด



เราสามารถใช้ Vertex Shader ทำอะไรได้บ้าง?

- Skinning

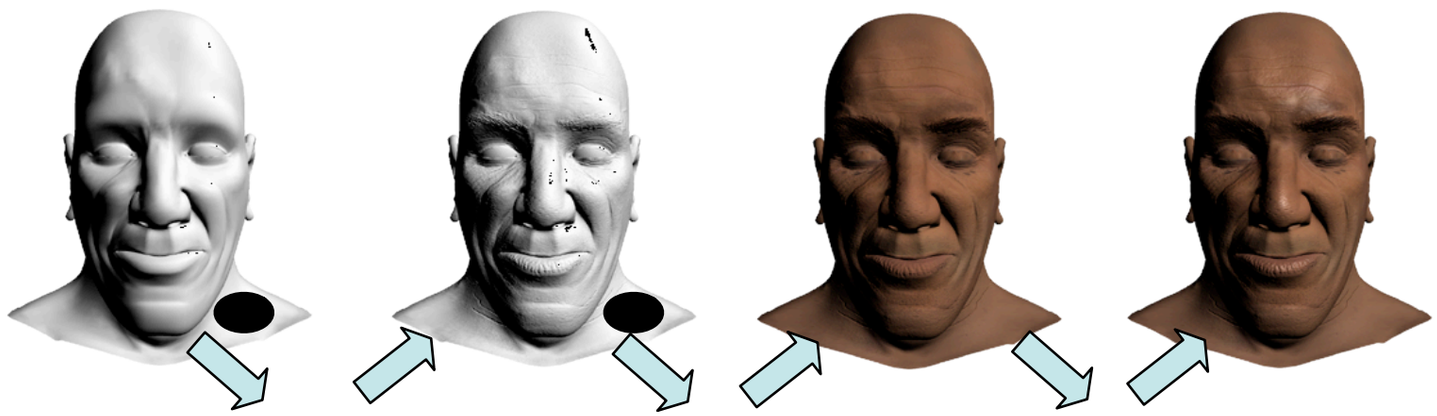


Dolphin #1 **Morphed Dolphin** **Dolphin #2**

Images courtesy of Microsoft

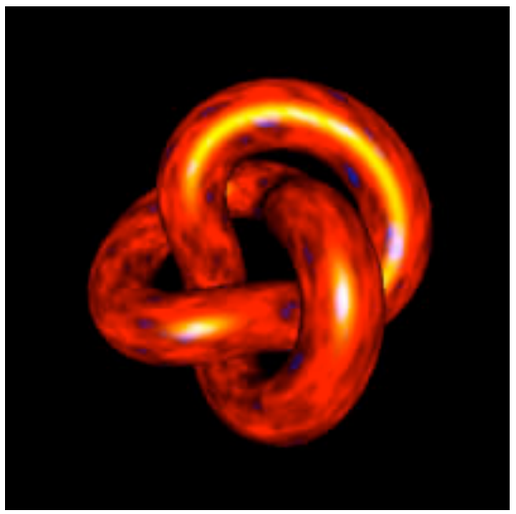
Fragment Shader

- คำนวณสีของ fragment
- รับข้อมูลที่ interpolate แล้วจาก vertex shader
- อ่านข้อมูลเพิ่มเติมจาก texture หรือตัวแปรอื่นๆ



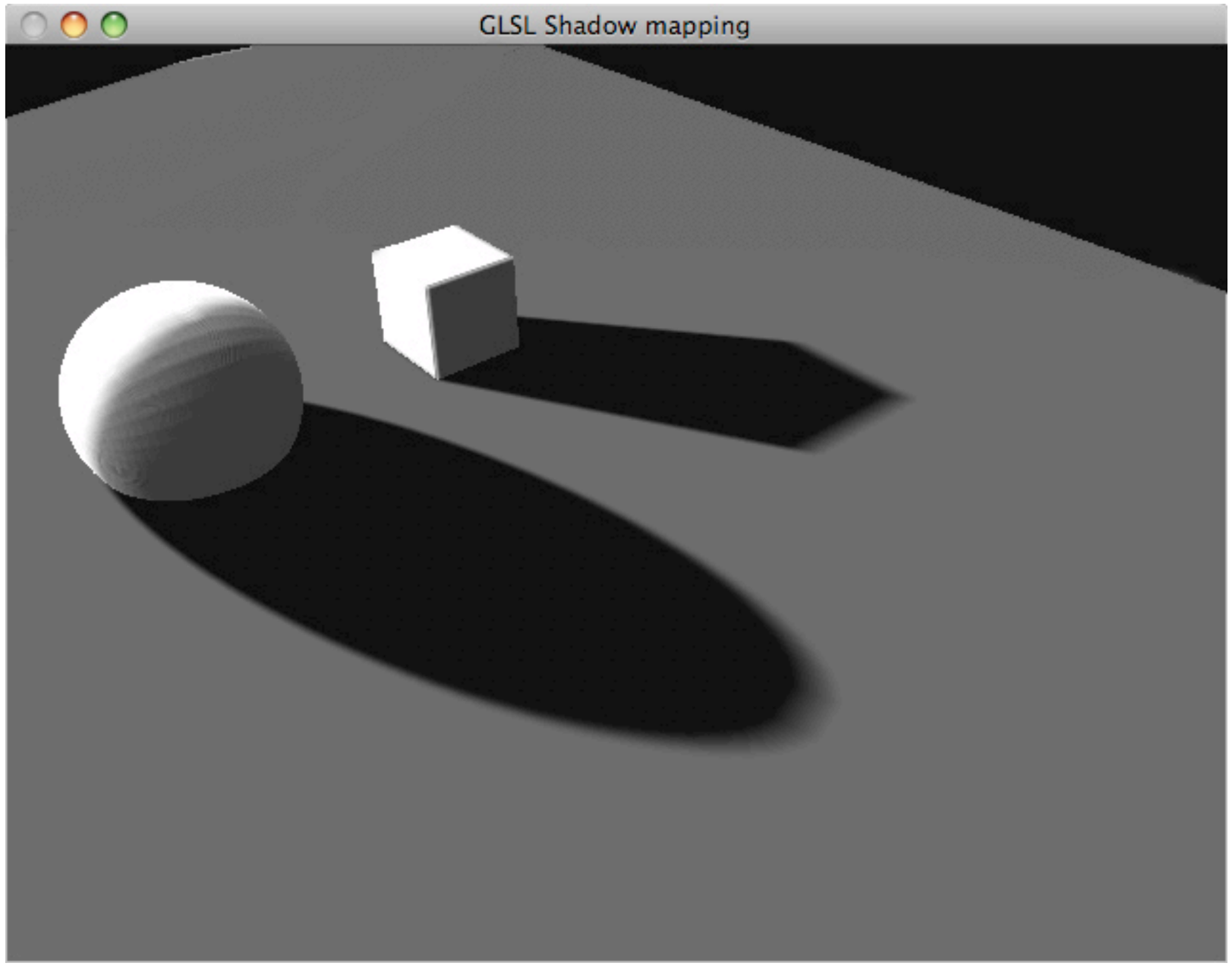
เราสามารถใช้ Fragment Shader ทำอะไรได้บ้าง?

- Material แปลกๆ
  - ผิวมันวาว
  - สะท้อนแสง หักเหแสง
  - ผิวขรุขระ ตะปุ่มตะป่ำ
  - ลายไม้



เราสามารถใช้ Fragment Shader ทำอะไรได้บ้าง?

- เงา

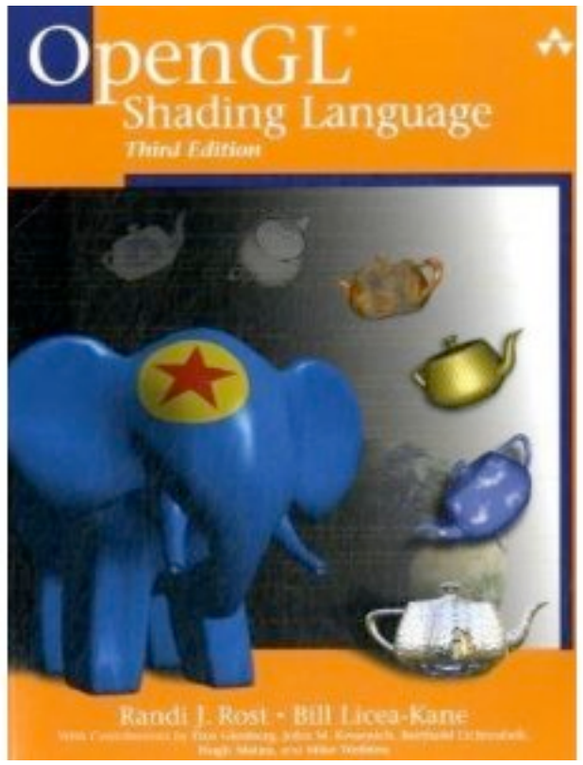


http://www.fabiensanglard.net/shadowmappingPCF/8x8kernel_nVidia.jpg

ภาษา GLSL

ภาษา GLSL

- ลักษณะคล้ายกับภาษา C/C++
- ใช้สำหรับเขียน shader สำหรับรันบน GPU
  - vertex, fragment, geometry
  - แต่ในวิชานี้จะพูดถึงแค่ vertex กับ fragment เท่านั้น
- มีพื้นฐานมาจาก OpenGL
- มีใช้ครั้งแรกใน OpenGL เวอร์ชัน 2.0 (2004)



GLSL Program

- ณ เวลาหนึ่ง GPU จะรันโปรแกรมอยู่หนึ่งโปรแกรม
  - เริ่มต้น OpenGL จะใช้โปรแกรมของมันเอง ซึ่งทำงานตาม OpenGL graphics pipeline
  - เราสามารถบอกให้ OpenGL เปลี่ยนมาใช้โปรแกรมที่เราสร้างขึ้นได้
  - หลังจากนั้นเราสามารถบอกให้ OpenGL กลับไปใช้ graphics pipeline ตามเดิมได้

GLSL Program

- โปรแกรมหนึ่งๆ จะประกอบด้วย shader หลายๆ ตัว
  - บางตัวเป็น vertex shader
  - บางตัวเป็น fragment shader
  - ไม่จำเป็นมีทั้ง vertex และ fragment shader
  - แต่ควรจะมีไว้ทั้งสองตัว เพราะเราจะได้รู้พฤติกรรมของมันชัดเจน

Shader

- ซอร์สโค้ดของ shader มีลักษณะคล้ายภาษา C และ C++
  - มีชนิดข้อมูล, นิพจน์ทางคณิตศาสตร์, คำสั่งควบคุม คล้ายๆ กัน
  - สามารถแบ่งโปรแกรมออกเป็นฟังก์ชัน
- จุดเริ่มต้นการทำงานอยู่ในที่ฟังก์ชัน void main( )
  - ไม่ใช่ int main(int argc, char **argv) เหมือนภาษา C ปกติ

Shader

- อาจ vertex shader หรือ fragment หนึ่งๆ อาจถูกแบ่งซอร์สโค้ดของ shader ออกเป็นหลายๆ ไฟล์
- แต่ในหลายๆ ไฟล์นั้นจะต้องมีฟังก์ชัน void main( ) เพียงแค่ฟังก์ชันเดียว
- โปรแกรมที่มีทั้ง vertex shader และ fragment shader จะมีฟังก์ชัน main สองตัว
  - ตัวหนึ่งอยู่ใน vertex shader
  - ตัวหนึ่งอยู่ใน fragment shader

โครงสร้าง Shader

```
/*
Multiple-lined comment
*/

// Single-lined comment

//
// Global variable definitions
//

void main()
{
    //
    // Function body
    //
}
```

โปรแกรมวาดสามเหลี่ยมสีเขียว

Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
```

การทำงาน

- สมมติว่าเราทำการสร้างโปรแกรมเสร็จเรียบร้อยแล้ว
- แล้วบอกให้ OpenGL ใช้โปรแกรมที่เราสร้างขึ้นมา
- สมมติในการวาดภาพโดยใช้ OpenGL เรามีการสั่งวาดสามเหลี่ยมดังนี้
-
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

การทำงานของ Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

โค้ด GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

โค้ด OpenGL

การทำงานของ Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

โค้ด GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

โค้ด OpenGL

gl_ModelViewProjectionMatrix
เป็นตัวแปรพิเศษที่มีค่าเท่ากับ
ผลคูณของ projection matrix
กับ modelview matrix

การทำงานของ Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

โค้ด GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

โค้ด OpenGL

gl_Vertex
เป็นตัวแปรพิเศษที่มีค่าเท่ากับ
ค่าที่เรากำหนดให้ด้วยคำสั่ง
glVertex
(กล่าวคือตำแหน่งใน
object space **นั่นเอง)**

การทำงานของ Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

โค้ด GLSL

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(-1, 1, -1, 1);

glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

glBegin(GL_TRIANGLES);
glVertex3f(-0.5, -0.5, 0);
glVertex3f( 0.5, -0.5, 0);
glVertex3f(   0,  0.5, 0);
glEnd();
```

โค้ด OpenGL

เนื่องจากใน OpenGL
มีการเรียก glVertex3f 3 ครั้ง
vertex program ข้างบน
จะถูกเรียก 3 ครั้ง
ครั้งแรก gl_Vertex = (-0.5, -0.5, 0, 1)
ครั้งที่สอง gl_Vertex = (0.5, -0.5, 0, 1)
ครั้งที่สาม gl_Vertex = (0, 0.5, 0, 1)

การทำงานของ Vertex Shader

```
void main()
{
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

โค้ด GLSL

gl_Position เป็นตัวแปรพิเศษสำหรับ
เก็บตำแหน่งของ vertex ใน clip space
(หลังจากคูณด้วย modelview และ projection matrix ด้วย)

เนื่องจาก vertex shader ต้องกำหนดตำแหน่งใน clip space
มันจะต้องทำการเซตค่าให้ gl_Position เสมอ

Vertex shader นี้จึงไม่ได้ทำอะไรต่างจากที่ OpenGL ทำอยู่แล้ว

การทำงานของ Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
```

โค้ด GLSL

การทำงานของ Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
```

โค้ด GLSL

gl_FragColor **เป็นตัวแปรพิเศษสำหรับเก็บสีของ** fragment

เนื่องจาก fragment shader **มีหน้าที่กำหนดสีของ** fragment **มัน จะต้องเซตค่า** gl_FragColor **เสมอ**

การทำงานของ Fragment Shader

```
void main()
{
    gl_FragColor = vec4(0,1,0,1);
}
```

โค้ด GLSL

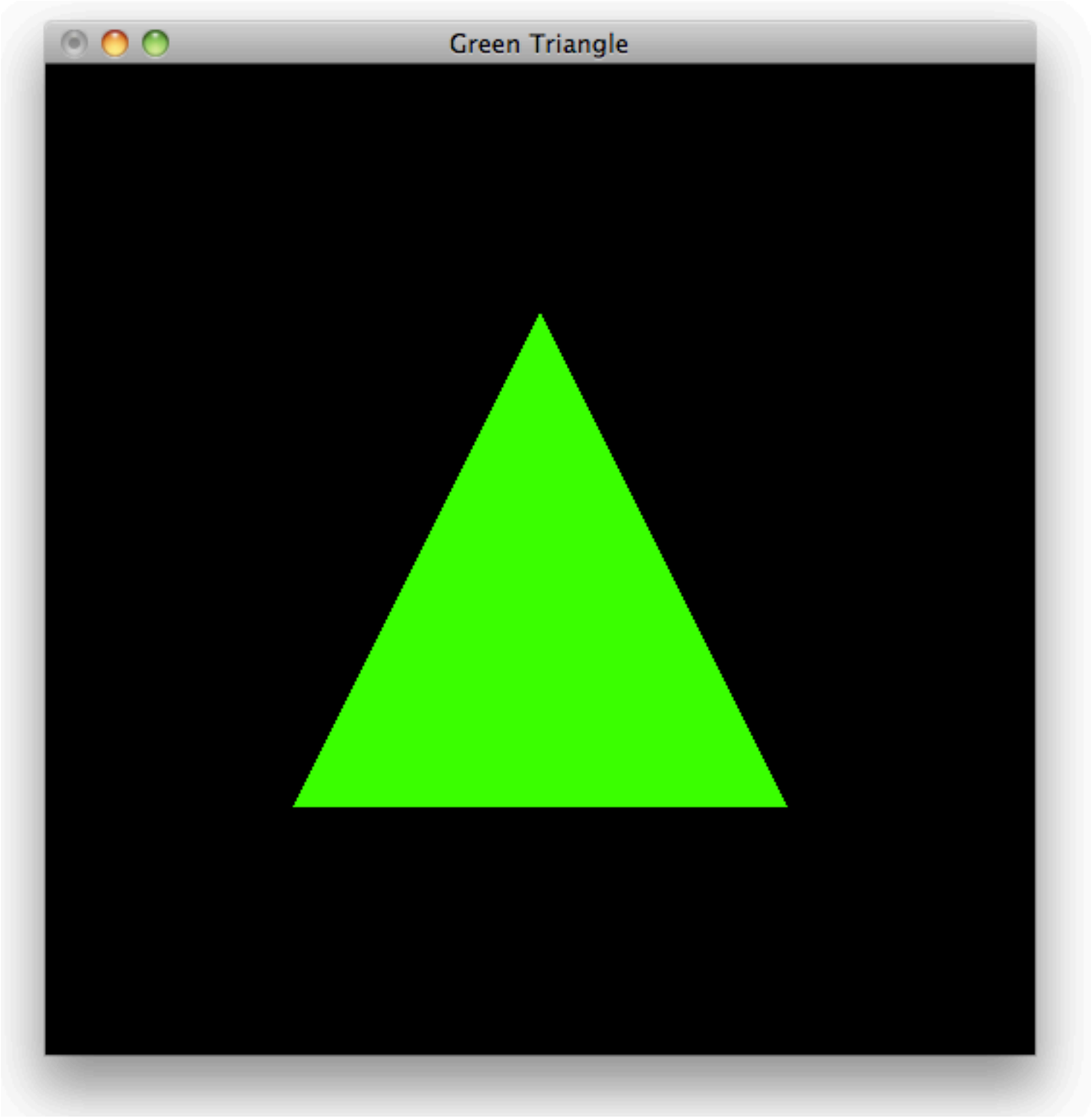
vec4 เป็นชนิดของตัวแปรสำหรับเก็บเวกเตอร์สี่มิติ

vec4 สามารถใช้เก็บ

- homogeneous coordinate
- สี RGBA

vec4(0,1,0,1) เป็นการสร้างเวกเตอร์สี่มิติแทนสี RGBA ที่มี R=0, G=1, B=0, A=1 ซึ่งก็คือสีเขียว

ผลลัพธ์



API ภาษา C ที่เกี่ยวข้องกับ GLSL

GLSL กับภาษา C

- GLSL เป็นส่วนหนึ่งของ OpenGL
- OpenGL เป็น API ที่เขียนในภาษา C สำหรับใช้ติดต่อกับ GPU
- ดังนั้นการเรียกใช้ GLSL จึงต้องทำผ่านฟังก์ชันภาษา C ที่ติดมากับ OpenGL
- OpenGL ไม่ได้เตรียมโปรแกรมสำหรับคอมไพล์ GLSL มาให้
  - แต่มีฟังก์ชันใน OpenGL ที่ใช้สำหรับคอมไพล์ GLSL โดยเฉพาะ
- เราต้องเขียนโค้ดภาษา C สำหรับคอมไพล์และสร้างโปรแกรมภาษา GLSL เอง

GLEW

- ภาษา GLSL มากับ OpenGL 2.0
- แต่ไลบรารี OpenGL ที่มากับระบบปฏิบัติการอาจจะเก่ากว่านั้น
    - เช่น ใน Windows จะมี OpenGL 1.1
- ดังนั้นหากอยากใช้ GLSL จะต้องมี GLEW
- รายละเอียดเพิ่มเติมดูได้ในสไลด์ของคาบที่แล้ว

การสร้างและเรียกใช้โปรแกรม GLSL ผ่าน C

- มี 7 ขั้นตอน
    1. สร้าง shader object
    2. เอา source code ของ shader ป้อนให้ shader object
    3. ทำการคอมไพล์ shader
    4. สร้าง program object
    5. เอา shader ไปติดกับโปรแกรม
    6. ลิงก์โปรแกรม
    7. บอกให้ OpenGL ใช้โปรแกรมแทนการใช้ graphics pipeline

1. สร้าง Shader Object

- Shader object คือโครงสร้างข้อมูลที่ OpenGL ใช้เก็บรายละเอียดของ shader ที่เขียนด้วยภาษา GLSL ตัวหนึ่ง

- สร้างด้วยฟังก์ชัน
  GLuint **glCreateShader**(GLenum *shaderType*)

- shaderType มีค่าได้สองแบบคือ
  GL_VERTEX_SHADER และ GL_FRAGMENT_SHADER

- คืน “หมายเลข” ของ shader ที่สร้างขึ้นมากลับให้ผู้เรียก

- หมายเลขนี้จะเอาไปใช้อ้างถึง shader ตัวนี้ในอนาคต

1. สร้าง Shader Object

- ตัวอย่าง

```
GLuint vert_id;
GLuint frag_id;

vert_id = glCreateShader(GL_VERTEX_SHADER);
frag_id = glCreateShader(GL_FRAGMENT_SHADER);
```

- ในโค้ดข้างบนเราสร้าง shader สองตัว
  - ตัวหนึ่งเป็น vertex shader แล้วเก็บหมายเลขของมันไว้ในตัวแปร vert_id
  - อีกตัวหนึ่งเป็น fragment shader แล้วเก็บหมายเลขของมันไว้ในตัวแปร frag_id

2. กำหนดซอร์สโค้ดของ shader

- กำหนดโดยฟังก์ชัน
  void **glShaderSource**(GLuint *shader*,
  GLsizei *count*,
  const GLchar ***string*,
  const GLint **length*);

- shader คือหมายเลขของ shader object ที่เราต้องการกำหนดซอร์สโค้ด
- ฟังก์ชันนี้รับซอร์สโค้ดเป็นอะเรย์ของสตริงซึ่ง OpenGL จะเอามาต่อกันโดยอัตโนมัติ
- ตัวซอร์สโค้ดอยู่ในอะเรย์ string (สังเกตว่ามันเป็นอะเรย์ของอะเรย์ของ char)
- ความยาวของสตริงแต่ละตัวใน string อยู่ในอะเรย์ length
- count คือความยาวของอะเรย์ string และ length (กล่าวคือมีสตริงกี่ตัว)
- ถ้า length เป็น NULL หมายความว่าสตริงทั้งหมดใน string เป็น null-terminated string (จบด้วยรหัสตัวอักษร \0)

2. กำหนดซอร์สโค้ดของ shader

- หากจะนำซอร์สโค้ดที่เราเห็นในสไลด์ก่อนๆ มาใช้ได้ เราต้องแปลงมันเป็นสตริงในภาษา C เสียก่อน

- ทางเลือกสองทาง
  - อ่านจากไฟล์อื่นๆ
  - เขียนใส่ในไฟล์ภาษา C เอาเอง

- ผมเลือกใช้ทางเลือกที่สอง
  - เวลารันโปรแกรมจริงๆ เราอาจไม่รู้ว่าไฟล์ซอร์สโค้ดอยู่ที่ไหนกันแน่
  - ทางเลือกที่สองทำให้เราสามารถเขียนไลบรารีที่มี shader ได้
  - เราไม่จำเป็นต้องเขียนสตริงเอง แต่เขียนโปรแกรมให้เขียนสตริงแทนเราได้

2. กำหนดซอร์สโค้ดของ shader

- ตัวอย่าง

```
const char * vert_code = "\
void main()\n\
{\n\
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;\n\
}\n\
";

const char * frag_code = "\
void main()\n\
{\n\
    gl_FragColor = vec4(0,1,0,1);\n\
}\n\
";

glShaderSource(vert_id, 1, &vert_code, NULL);
glShaderSource(frag_id, 1, &frag_code, NULL);
```

3. **คอมไพล์** Shader

- โดยใช้ฟังก์ชัน
  void **glCompileShader**(GLuint *shader*);

  - shader คือหมายเลขของ shader ที่ต้องการ compile

- ตัวอย่าง

```
glCompileShader(vert_id);
glCompileShader(frag_id);
```

ตรวจสอบว่าโค้ดคอมไพล์ผ่านหรือไม่

- เช่นเดียวกับการเขียนโค้ดภาษาอื่น
  เรามีโอกาสจะเขียนโค้ดแล้วผิดไวยากรณ์ของภาษา GLSL

- เราสามารถอ่าน error message จากการคอมไพล์โดยใช้ฟังก์ชัน
  glGetShaderInfoLog

- อย่างไรก็ดี รายละเอียดการใช้ฟังก์ชันนี้ยุ่งยาก
  ผมจึงจะเสนอโค้ดสำหรับพิมพ์ error message ออกมาเลยในสไลด์ต่อไป

ตรวจสอบว่าโค้ดคอมไพล์ผ่านหรือไม่

- ```
  void print_shader_info_log(GLuint obj)
  {
      int infologLength = 0;
      int charsWritten  = 0;
      char *infoLog;

      glGetShaderiv(obj, GL_INFO_LOG_LENGTH,&infologLength);

      if (infologLength > 0)
      {
          infoLog = (char *)malloc(infologLength);
          glGetShaderInfoLog(obj, infologLength, &charsWritten, infoLog);
          printf("%s\n",infoLog);
          free(infoLog);
      }
  }
  ```

- โค้ดนี้ไปลอกมาจาก http://www.lighthouse3d.com/opengl/glsl/index.php?
  ```

ตรวจสอบว่าโค้ดคอมไพล์ผ่านหรือไม่

- ให้เรียก **print_shader_info_log** ทันทีหลังจากเรียก **glCompileShader** แล้ว

- ตัวอย่าง

```
glCompileShader(vert_id);
glCompileShader(frag_id);

print_shader_info_log(vert_id);
print_shader_info_log(frag_id);
```

4. สร้าง Program object

- Program object คือโครงสร้างข้อมูลที่ OpenGL ใช้เก็บข้อมูลของโปรแกรมภาษา GLSL หนึ่งโปรแกรม

- สร้างโดยใช้ฟังก์ชัน
  GLuint **glCreateProgram**(void);
  - คืนหมายเลขของโปรแกรมใหม่ ซึ่งเราสามารถใช้ในการอ้างถึงมันในภายหลังได้

- ตัวอย่าง

```
GLuint prog_id;

prog_id = glCreateProgram();
```

5. เอา shader ไปติดกับโปรแกรม

- ด้วยคำสั่ง
  void **glAttachShader**(GLuint *program*, GLuint *shader*);

  - program คือหมายเลขของโปรแกรม
  - shader คือหมายเลขของ shader

- ตัวอย่าง

```
glAttachShader(prog_id, vert_id);
glAttachShader(prog_id, frag_id);
```

6. ลิงก์โปรแกรม

- การ link คือการเอาส่วนประกอบของโปรแกรมหลายๆ ส่วนมาเชื่อมต่อกัน

- ทำด้วยคำสั่ง
  void **glLinkProgram**(GLuint *program*);

- ตัวอย่าง

```
glLinkProgram(prog_id);
```

ตรวจสอบว่าการลิงก์มีปัญหาหรือไม่

- ด้วยฟังก์ชัน print_program_info_log ต่อไปนี้

```
void print_program_info_log(GLuint obj)
{
    int infologLength = 0;
    int charsWritten  = 0;
    char *infoLog;
    glGetProgramiv(obj, GL_INFO_LOG_LENGTH,&infologLength);
    if (infologLength > 0)
    {
        infoLog = (char *)malloc(infologLength);
        glGetProgramInfoLog(obj, infologLength, &charsWritten, infoLog);
        printf("%s\n",infoLog);
        free(infoLog);
    }
}
```

- โค้ดนี้ไปลอกมาจาก http://www.lighthouse3d.com/opengl/glsl/index.php? เช่นเคย

ตรวจสอบว่าการลิงก์มีปัญหาหรือไม่

- ให้เรียก **print_program_info_log** หลังจากเรียก **glLinkProgram** ทันที

- ตัวอย่าง

```
glLinkProgram(prog_id);
print_program_info_log(prog_id);
```

7. บอก OpenGL ให้ใช้โปรแกรมที่เราสร้าง

- ด้วยคำสั่ง
  void **glUseProgram**(GLuint *program*);

- ตัวอย่าง

```
glUseProgram(prog_id);
```

โค้ดการสร้างและใช้โปรแกรม GLSL ทั้งหมด

```
vert_id = glCreateShader(GL_VERTEX_SHADER);
frag_id = glCreateShader(GL_FRAGMENT_SHADER);

glShaderSource(vert_id, 1, &vert_code, NULL);
glShaderSource(frag_id, 1, &frag_code, NULL);

glCompileShader(vert_id);
glCompileShader(frag_id);

print_shader_info_log(vert_id);
print_shader_info_log(frag_id);

prog_id = glCreateProgram();
glAttachShader(prog_id, vert_id);
glAttachShader(prog_id, frag_id);

glLinkProgram(prog_id);
print_program_info_log(prog_id);

glUseProgram(prog_id);
```

เครื่องมือสำหรับคอมไพล์โค้ดตัวอย่าง

CMake และ ritsu

- โค้ดตัวอย่างที่ให้ในชั้นเรียนจะใช้เครื่องมือ 2 ชิ้นในการคอมไพล์
  - CMake (http://www.cmake.org/)
  - ritsu (http://github.com/dragonmeteor/ritsu)

ติดตั้ง CMake

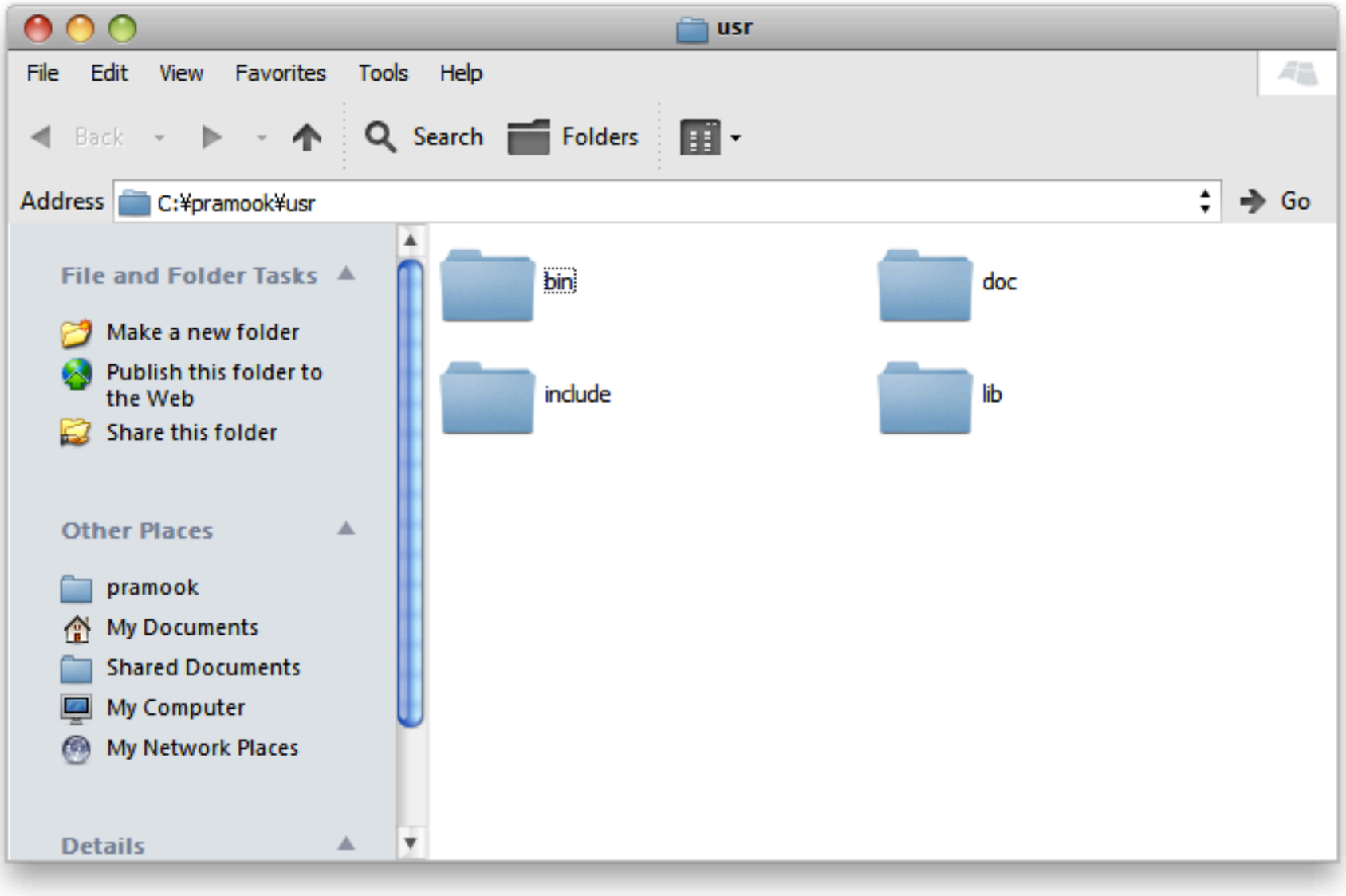
- ดาวน์โหลด installer ของ CMake เวอร์ชันใหม่ล่าสุดจาก http://www.cmake.org/cmake/resources/software.html

- ดูที่ “Binary Distribution”

ติดตั้ง Library ที่จำเป็น

- สร้าง directory ไว้หนึ่ง directory ที่ไหนก็ได้ สำหรับเก็บ library เช่น GLUT, GLEW, และ DevIL
- ของผมใช้ C:\Pramook\usr
- ในไดเรคทอรีนั้นให้สร้าง directory ย่อยสามตัว
  - bin สำหรับเก็บไฟล์ .dll และ .exe
  - lib สำหรับเก็บไฟล์ .lib
  - include สำหรับเก็บไฟล์ .h

ติดตั้ง Library ที่จำเป็น

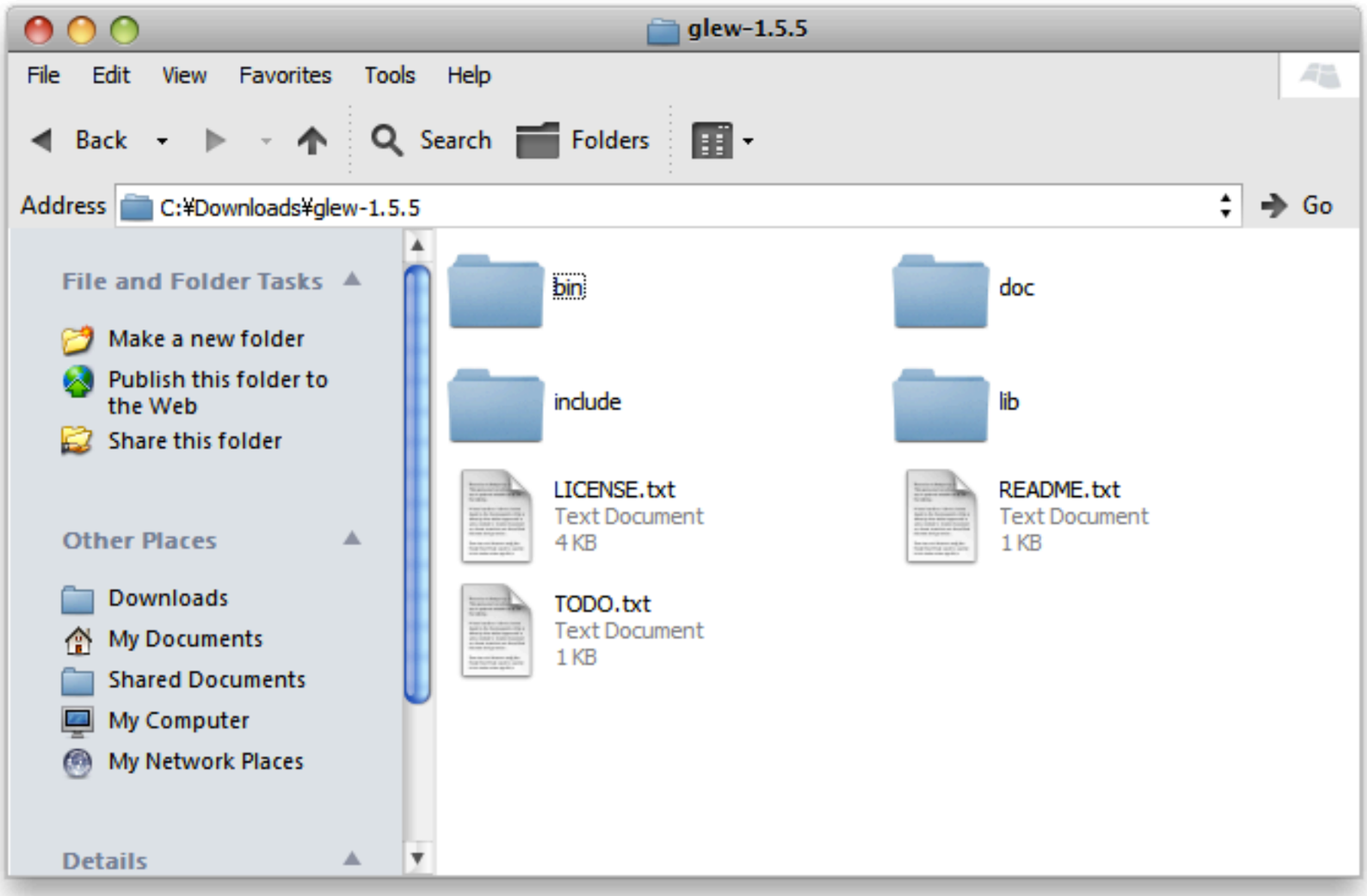


ติดตั้ง Library ที่จำเป็น

- ติดตั้ง Library สามตัว
    - GLUT จาก http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip
    - GLEW จาก http://glew.sourceforge.net/
        - ดาว์นโหลดเวอร์ชัน Windows 32-bit หรือ 64-bit ตามระบบปฏิบัติการ
- DevIL จาก http://openil.sourceforge.net/download.php
    - เลือก DevIL 1.7.8 SDK for 32-bit หรือ 64-bit Windows ตามระบบปฏิบัติการที่คุณใช้

ตัวอย่าง: การลง GLEW

- เมื่อแตกไฟล์ glew-1.5.5-win32.zip ที่ดาวน์โหลดจาก glew.sourceforge.net มาจะได้ directory ชื่อ glew-1.5.5



ตัวอย่าง: การลง GLEW

- เนื่องจากโครงสร้าง directory ใน ZIP ไฟล์ ตรงกับที่วางไว้อยู่แล้ว เราสามารถ:
  - copy directory “lib” ไปทับ “C:\pramook\usr\lib”
  - copy directory “bin” ไปทับ “C:\pramook\usr\bin”
  - copy directory “include” ไปทับ “C:\pramook\usr\include”

ข้อควรระวัง: การลง GLUT

- เมื่อแตกไฟล์ glut-3.7.6-bin จะพบไฟล์
  - glut32.dll --> เอาไปใส่ใน C:\pramook\usr\bin
  - glut32.lib --> เอาไปใส่ใน C:\pramook\usr\lib
  - glut32.lib --> กรณีพิเศษ!!!
- สำหรับ glut32.lib สร้าง directory “C:\pramook\usr\include\GL” แล้วเอามันไปใส่ในนั้น
  - อย่าเอาไปใส่ใน C:\pramook\usr\include โดยตรง

การติดตั้ง ritsu

- ให้ติดตั้งภาษา Ruby ก่อน (ritsu เขียนขึ้นด้วยภาษา Ruby)
  - ดาวน์โหลดจาก http://www.ruby-lang.org/en/downloads/
  - ดาวน์โหลด installer ของเวอร์ชันไหนก็ได้ตั้งแต่ 1.8.7 เป็นต้นไป
- หลังจากติดตั้งเสร็จแล้วให้
  - ตั้งตัวแปร PATH ให้มีไดเรกทอรี C:\ruby\bin อยู่ภายใน
  - ทดสอบว่าติดตั้งถูกต้องไหมโดยเปิด cmd แล้วพิมพ์ irb แล้วกด enter

การติดตั้ง ritsu

- เปิด command line (คำสั่ง cmd)
- สั่ง
    - gem update --system
    - gem install ritsu

แก้ไข Environment Variable

- CMAKE_PREFIX_PATH
  - ให้มีค่าเท่ากับไดเรคทอรีที่คุณใช้เก็บไลบรารีต่างๆ
  - ในกรณีของผมคือ C:\pramook\usr

คอมไพล์โค้ดตัวอย่าง

- ดาวน์โหลดโค้ดตัวอย่างจากเว็บไซต์วิชา
- สมมติว่า download โค้ดตัวอย่างของ Lecture 01 มา (lecture01.zip)
- สมมติว่าแตก lecture01.zip ลงใน directory C:\lecture01
- ให้เปิด command-prompt (คำสั่ง cmd)
- เปลี่ยน directory เข้าไปใน directory ของโค้ดตัวอย่าง
  - cd C:\lecture01
- สั่ง
  - thor lecture01:update
- เปิดไดเรคทอรี C:\lecture01\build แล้วจะเจอไฟล์ lecture01.sln อยู่ในนั้น
- เปิด lecture01.sln ด้วย Visual Studio แล้วคอมไพล์ตามปกติ