Name Solution

GPU Programming
EE 4702-1
Midterm Examination

Wednesday, 12 November 2014 9:30–10:20 CST

Problem 1		(18 pts)
Problem 2		(12 pts)
Problem 3		(24 pts)
Problem 4		(12 pts)
Problem 5		(14 pts)
Problem 6		(20 pts)
Exam Total		(100 pts)

Alias Phyloe has landed!

Good Luck!

Problem 1: [18 pts] Appearing below is a simplified version of the vertex shader `vs_main` from Homework 4. One way to reduce the amount of computation it must perform is by pre-computing values and making them available as `vs_main` inputs. For example, rather than compute `radial_idx` each time, it can instead be pre-computed and stored in the `w` component of `gl_Vertex` or it can be stored in a user-defined vertex shader input named `radial_idx`.

```
void vs_main() {
  int bidx = gl_Vertex.x;              int ti = gl_Vertex.y;

  int radial_idx = bidx * opt_segments + ti;
  float theta = (1.0/opt_segments) * radial_idx * omega;

  vec3 pos1 = balls_pos[bidx-1].xyz;    vec3 pos2 = balls_pos[bidx].xyz;
  vec3 v12 = pos2.xyz - pos1.xyz;
  vec3 p = pos1 + float(ti) / opt_segments * v12;

  vec3 va = spiral_radius * normalize( vec3(0,v12.z,-v12.y) );
  vec3 vb = spiral_radius * normalize( cross(v12,va) );

  vec3 radial = va * cos(theta) + vb * sin(theta);
  vec3 p_outer = p + radial;           vec3 p_inner = p + 0.5 * radial;
```

Consider two candidates for new vertex shader inputs: `va/vb` (count these as one candidate) and `theta`.

(*a*) Estimate the amount by which each candidate would reduce the amount of computation to be performed **by the vertex shader**.

☑ Computation reduction by making `va/vb` an input:

☑ Computation reduction by making `theta` an input:

To solve this problem we need to estimate the amount of computation performed for each candidate. As done in class we will estimate computation by tallying the number of arithmetic operations needed.

The easier computation is for `theta`. Look at the line that assigns `theta` and the line that assigns `radial_idx`. They perform three multiplies, a divide, and an addition. These computations would not be necessary if `theta` were an input to the shader.

For `va` first consider the normalize. With optimization that recognizes that the x component is zero, that would be two multiplies and an add to compute $l^2 = y^2 + z^2$. A reciprocal square root and two multiplies are needed to compute $(0, yl^{-1}, zl^{-1})$, and two more multiplies are needed for `spiral_radius`. The total is 6 multiplies, an add, and a reciprocal square root (to compute $l^{-1} = 1/\sqrt{y^2+z^2}$). For `vb` we need to do a cross product which is 6 multiplies and four adds, plus 6 more multiply/adds and a reciprocal square root for the normalize and `spiral_radius` multiplication. The total for `va` and `vb` is 24 multiplies or multiply/adds, 3 adds and two reciprical square roots.

Grading Note: Many students computed the amount of data rather than the amount of computation.

(*b*) One of the candidates would result in a large increase in CPU to GPU data transfer. Which one is it, and why?

☑ Candidate causing a big increase in CPU to GPU data transfer: Variables `va` and `vb`.

☑ Reason for **big** increase:

Variables `va` and `vb` will result in a big increase in CPU to GPU traffic because, unlike `theta`, they need to be changed every frame. Notice that `theta` only depends upon constants, such as `opt_segments`, and things that are always the same for a particular vertex, such as `bidx`. Therefore the value of `theta` for each vertex can be put in a buffer object and used for a new vertex shader

input. The buffer object would be transferred to the GPU just once. In contrast, `va` and `vb` depend on `balls_pos` which changes each frame.

(*c*) In the code above there is a much better candidate for a new vertex shader input other than `va/vb` or `theta`. It would reduce computation by a significant amount and would not have a big impact on data transfer. *Hint: it's not a variable, but an expression, or several expressions.*

☑ The much better candidate for an input attribute is:

Rather than put `theta` in a buffer object, which would only save a small amount of computation, put values for `sin(theta)` and `cos(theta)`. Like `theta`, these don't change from frame to frame, and unlike `theta` they save much more computation because `sin` and `cos` are expensive to compute.

Problem 2: [12 pts] Appearing below is the fragment shader from the solution to Homework 4. Recall that this shader does not render the spiral surface in places where the texture is dark, giving the appearance of holes.

(*a*) Modify the shader code so that the texture itself is only applied to the front side of the spiral, and so that the holes still appear on **both** sides. *Hint: Can be done with a line or two of code.*

☑ Modify so holes are on both sides, but texture just on front.

```
void fs_main() {
  vec4 color = is_edge ? gl_FrontMaterial.ambient :
        gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);

  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  // SOLUTION: Only use texel to color surface for non-edge front-facing prims.
  vec4 texel_apply = is_edge || !gl_FrontFacing ? vec4(1,1,1,1) : texel;

  gl_FragColor = texel_apply * generic_lighting(vertex_e, color, normalize(normal_e));

  gl_FragDepth = gl_FragCoord.z;
}
```

(*b*) The fragment shader code below, based on Homework 4, is supposed to use different colors for the front, back, and edge of the spiral. The code below has an error and the commented-out line shows the correct code. How will this error change the appearance of the spiral?

☑ With this error spiral will appear...

Short answer: ... with the exact same color everywhere, that is, there will be no shading based upon light location, etc.

Long answer: The lighting routine is not being applied. The lighting routine returns a color based on `our_color` but taking into account the location and orientation of the vertex (the other input parameters) and the location and color of the light (which are uniform variables). In the erroneous code none of this is being done, instead `our_color` is being used and so the color of the spiral will be exactly the same everywhere.

```
void fs_main() {
  vec4 our_color = is_edge ? gl_FrontMaterial.ambient :
            gl_FrontFacing ? gl_FrontMaterial.diffuse : gl_BackMaterial.diffuse;

  vec4 texel = is_edge ? vec4(1,1,1,1) : texture(tex_unit_0,gl_TexCoord[0].xy);
  bool hole = texel.r + texel.g + texel.b < 0.05;
  if ( hole ) discard;

  //  gl_FragColor = texel * generic_lighting(vertex_e, our_color, normalize(normal_e));
  gl_FragColor = texel * our_color;

  gl_FragDepth = gl_FragCoord.z;
}
```

Problem 3: [24 pts] Answer each question below.

(*a*) An object at point P_1 is moving at velocity v_1 just before collision with a wall with normal $\hat{w}$. The collision is completely elastic, meaning the object's speed (the magnitude of velocity) will not change. Write an expression for the velocity after the collision?

☑ Expression showing velocity after collision:

Short answer: $v_1 - 2\left(v_1 \cdot \hat{w}\right)\hat{w}$.

The speed by which the point is moving towards the wall is found by taking the dot product of the velocity with the wall normal: $v_1 \cdot \hat{w}$. The approach velocity is the vector $\left(v_1 \cdot \hat{w}\right)\hat{w}$. This velocity gets reversed by the collision, so the velocity after collision is $v_1 - 2\left(v_1 \cdot \hat{w}\right)\hat{w}$.

(*b*) An object at P_1 is connected to an object at P_2 by an ideal spring having a relaxed distance of d and a spring constant h. Write an expression for the force on P_1 due to this spring.

☑ Expression giving force on P_1 due to spring:

The distance between the points is $||\overrightarrow{P_1P_2}||$. The magnitude of the force due to the spring is given by $||\overrightarrow{P_1P_2}|| - d$. (A positive force value pulls the points towards each other.) The force on P_1 is $\left(||\overrightarrow{P_1P_2}|| - d\right)\widehat{P_1P_2}$, where $\widehat{P_1P_2}$ is vector $\overrightarrow{P_1P_2}$ normalized.

(*c*) Point P_1 in 3D space has homogeneous coordinate $(3, 4, 6, 0.5)$. What is the corresponding Cartesian (ordinary every day) coordinate?

☑ Cartesian coordinate is:

The Cartesian coordinate is $(3/0.5, 4/0.5, 6/0.5) = (6, 8, 12)$.

(*d*) Homogeneous coordinates make it possible to express certain transformations using matrix multiplication that would not be possible with ordinary Cartesian coordinates. Name one such transformation.

☑ Transformation that's **not** possible using matrix multiplication with Cartesian coordinates:

There were two covered in class, translation and projection. Either answer would be correct. A scale transformation can be done using a feature of homogeneous coordinates, but scale transformations can also be done for Cartesian coordinates, and so scale transformation is not a correct answer.

Problem 4: [12 pts] Answer each question below.

(*a*) Our user's eye is at location (x, y, z) in our coordinate system. How do we provide this information to OpenGL (using the compatibility profile)? Be specific.

☑ User's location given to OpenGL by ...

Short Answer: We specify a modelview matrix that translates point (x, y, z) to the origin.

Long Answer: The location of a user's eye is specified using the OpenGL modelview matrix. To provide this information we must specify a modelview matrix that moves the user's eye to the origin. Moving the user's eye to the origin only requires a translation transformation, but for the modelview matrix we also want to rotate the scene so that our user's computer monitor is facing the $-z$ direction.

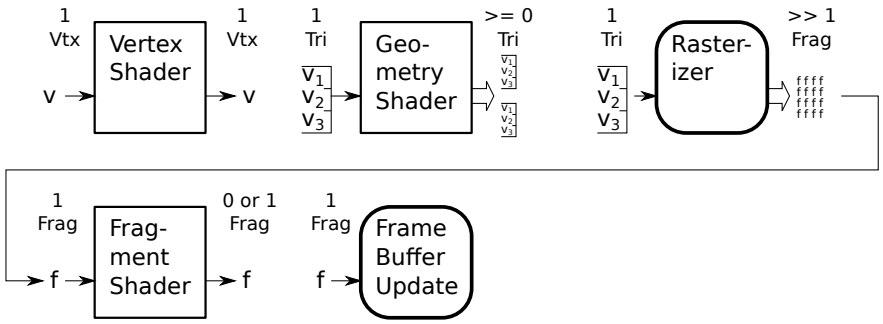
(*b*) The series of transformations performed by OpenGL (and shaders) maps 3D coordinates in our coordinate space to *window space* which are the 2D window coordinates corresponding to pixel locations. This transformation preserves coordinates' z component. Why is the z component needed, and how is it used?

☑ The z component is used for ...

The z component is used to decide which of two fragments located at the same pixel should be discarded (or how they should be blended together). Normally, the fragment further from the user would be discarded, but OpenGL allows for other possibilities.

This processes occurs after the fragment shader stage. The z component of the fragment to be written is compared to the z component already written into the frame buffer. If the fragment passes this depth test (the comparison) it is written, replacing (or blending with) the fragment that is already there (effectively discarding the old fragment). If the fragment fails the depth test it itself is discarded.

Problem 5: [14 pts] The diagram below shows a simplified form of the OpenGL rendering pipeline.



(*a*) The diagram shows 16 f's at the output of the rasterizer stage, but that's not necessarily the actual number.

☑ What does the actual number of f's depend on?

The number of f's depends upon the *projected* size of the triangle (the size of the triangle on the user's screen), and the resolution of the user's screen.

☑ How could one increase the number of f's without changing the input, $(V_1V_2V_3)$.

One could change the modelview matrix so that the user is closer to the triangle. Another option would be to increase the size of the window or render to a higher resolution device.

Grading Note: For some reason, most people answered "get a higher resolution monitor." Does something about the test remind people of Christmas?

(*b*) The boxes with rounded corners are part of what's called the *fixed functionality*. What does that mean, and why are those stages part of it.

☑ Fixed functionality means ...

The fixed functionality parts of the pipeline are parts that cannot be programmed. The Kronos Group (the organization defining the OpenGL standard) made those parts fixed function for performance reasons.

☑ The Rasterizer and Frame Buffer Update are fixed functionality because ...

... hardware is capable of performing the respective function better than software. Therefore making it programmable might improve flexibility, but it would slow down performance.

Problem 6: [20 pts] Answer each question below.

(*a*) The code below specifies vertices using a triangle strip. Modify the code so that it renders the same triangles using individual triangles.

☑ Modify code to use individual triangles ☑ without changing what's rendered.

```
glBegin(GL_TRIANGLES);

for ( int i=0   ;    i<num_v-2   ;    i++    )
  {

    glVertex3fv( coord[i] );
    glVertex3fv( coord[i+1] );
    glVertex3fv( coord[i+2] );

  }
glEnd();
```

(*b*) There is an important difference between specifying a color using `glColor` and `glMaterial`. One of them allows each vertex to have its own color, the other allows different properties to be set, such as ambient, diffuse, and specular but all vertices in a rendering pass must share these colors. *Note: This question applies only to the compatibility profile.*

☑ The command that allows each vertex to have its own color is:

The command is `glColor`, which specifies a vertex attribute.

☑ Why can't each vertex have its own set of color properties using either of these commands?

To limit the amount of data needed for each vertex. If each vertex *could* have its own full set of colors that would require more than eight 4-element vectors (values for the four types of material properties for each of two sides).

(*c*) Describe how the `GL_LINEAR_MIPMAP_LINEAR` minification method computes a filtered texel. Indicate how many texels are read from texture images, which images are read, and how they are combined.

☑ Number of texture images read:

An original texture image is used to construct a series of MIPMAP layers, with successively smaller widths, down to a width of one pixel. For `GL_LINEAR_MIPMAP_LINEAR` filtering two layers (texture images) are read, the two whose texels are closest in size to the pixel to be written.

☑ Number of texels read:

Four texels are read from each layer, for a total of eight texels.

☑ How they are combined:

A weighted average of the texel values is computed. The weight is based on how much of the texel covers the pixel to be written.