The Honor Code is the University's statement on academic integrity written by students in 1921. It articulates University expectations of students and faculty in establishing and maintaining the highest standards in academic work:

1. The Honor Code is an undertaking of the students, individually and collectively:
   a. that they will not give or receive aid in examinations; that they will not give or receive unpermitted aid in class work, in the preparation of reports, or in any other work that is to be used by the instructor as the basis of grading;
   b. that they will do their share and take an active part in seeing to it that others as well as themselves uphold the spirit and letter of the Honor Code.

2. The faculty on its part manifests its confidence in the honor of its students by refraining from proctoring examinations and from taking unusual and unreasonable precautions to prevent the forms of dishonesty mentioned above. The faculty will also avoid, as far as practicable, academic procedures that create temptations to violate the Honor Code.

3. While the faculty alone has the right and obligation to set academic requirements, the students and faculty will work together to establish optimal conditions for honorable academic work.

I acknowledge and accept the Honor Code.

NAME (Please Print):

Signature:

Note: This is exam is open-book, open-notes, open-laptop, but closed-network.

The exam consists of 5 questions. Each question is worth 20 points. Please answer all the questions in the space provided, overflowing on to the back of the page if necessary.

This exam has been designed to take 1 1/2 hr. However, you have 3 hours to complete the exam.
1. General (20 points)

A) [3 points] What is double buffering, and why is it important in animation?

*Double buffering is the use of separate draw (back) and display (front) buffers. The two are swapped only when drawing into the back buffer has been completed. This prevents the display of partially drawn frames, ensuring smooth, accurate frame transitions in animation.*

B) [5 points] Fill in the blanks: The axes in the HSV color space are called hue, saturation, and value. (1pt.) The cells in the human eye that are sensitive to color are called cones (1pt); they operate on the following axis/axes in HSV space: hue and saturation (1pt). The cells in the human eye which are sensitive to light and dark are called rods (1pt); they operate on the following axis/axes in HSV space: value (1pt).

- partial credit was given for “lightness” in place of “value”, “colorfulness” rather than “saturation”, &c.

C) [3 points] Fonts are usually represented by Bezier curves, but they can also be represented by bitmaps. What are the advantages of each approach?

*Many different answers were accepted, including:*
  
  **Bezier:** scale without artifacts. more efficient to store large fonts. intuitive to use for designers  
  **Bmp:** pixel by pixel control (esp. important at small scales). more efficient to store small or very complex fonts.

D) [3 points] Describe the relationship between a pixel and a fragment.

*Fragments are the result of breaking down geometry into single-pixel areas. They contain the necessary information (normals, etc.) to compute the color of that pixel. Because geometry may overlap, there are often more than one fragment per screen pixel.*

E) [6 points] Vectors and normals transform differently. Let \( \mathbf{v}' \) be the result of transforming a vector \( \mathbf{v} \). And let \( \mathbf{n}' \) be the result of transforming a normal \( \mathbf{n} \). Finally, suppose \( \mathbf{n} \) and \( \mathbf{v} \) have the same values for their coordinates; i.e. \( \mathbf{n} = \mathbf{v} = (a,b,c,0) \). For each of the following transforms, will \( \mathbf{v}' \) be the same or different than \( \mathbf{n}' \)? Mark each transform with the word “SAME” or “DIFFERENT”):

<table>
<thead>
<tr>
<th>Transform</th>
<th>same / different?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translate(2,2,2)</td>
<td>same</td>
</tr>
<tr>
<td>Scale(2,3,4)</td>
<td>different</td>
</tr>
<tr>
<td>RotateZ(45)</td>
<td>same</td>
</tr>
</tbody>
</table>
2. Signal processing and sampling (20 points)

A common operation in imaging is to resize an image. For example, we may want to convert a 1024 by 768 image to an 600 by 450 image to display it on a web page. There are two different cases to consider when resizing: (1) magnification, whereby the resolution is increased, and (2) minification, whereby the resolution is decreased.

In the following questions we start with a 512 by 512 image.

A) [10 points] The following function prototype is for a method which magnifies the image by a factor of 2 from 512 by 512 to 1024 by 1024:

```
// input has size (512,512), output has size (1024,1024)
magnify( STImage input, STImage output );
```

How might this function be implemented? Describe the best algorithm you can think of for magnifying an image (provide pseudocode below). Make sure you explain the details of any mathematical techniques you use. Then, compare your algorithm to an alternative algorithm and tell us why you think yours is better (that is, describe a potential artifact of the alternative algorithm and describe your approach to eliminating the artifact).

*One nice way to magnify an image is to use bilinear interpolation to set the values of the new pixels.*

```
magnify( STImage input, STImage output ) {
    for( int y=0; y < output.height; y++ ) {
        int iy = y/2;
        int jy = min(iy+1, input.height);
        for( int x = 0; x < output.width; x++ ) {
            int ix = x/2;
            int jx = min(ix+1,input.width);
            float v00 = input.GetPixel(ix,iy);
            float v01 = input.GetPixel(ix,jy);
            float v10 = input.GetPixel(jx,iy);
            float v11 = input.GetPixel(jx,jy);
            float fx = (x-ix)/2.0;
            float fy = (y-iy)/2.0;
            v = bilinear(fx,fy,v00,v01,v10,v11);
            output.SetPixel(x,y,v);
        }
    }
}
```
float bilinear(float x, float y, float v00, float v10, float v01, float v11) {
    float v0 = lerp(x,v00,v10);
    float v1 = lerp(x,v01,v11);
    return lerp(y,v0,v1);
}

Another way to magnify would be to use nearest neighbor interpolation. Nearest neighbor is not as good as bilinear interpolation because each input pixel will be expanded into a 2x2 block of pixels with the same value. The resulting image will look blocky. Bilinear interpolation will more smoothly interpolate between pixel values.

Grading

6 points for providing any reasonable algorithm that correctly interpolates values to form the new values.

4 points for presenting an alternative and stating why your algorithm is better.

A few people thought this algorithm had to do with compression. That’s not really correct, except to say that the output can be computed from the input and in some sense compresses well.

A few people also thought the artifacts were due to aliasing. Magnifying an image will not introduce aliasing since you have more samples. In general, the lower resolution input image will not have frequencies higher than what can be represented in the output image.
B [10 points] Here is a similar function prototype for minifying the image by a factor of 2 from 512 by 512 to 256 by 256:

```cpp
// input has size (512,512), output has size (256,256)
minify( STImage input, STImage output );
```

Algorithm 1. Suppose this function worked by simply throwing away ½ the pixels in the x and y directions by just choosing pixels in the image array whose x and y indices are even (the 0th, 2nd, ..., and 510th pixels).

What image artifact may result if you use this very simple algorithm? Will this artifact always be present? Under what conditions will the artifact be present, and when will they not be present?

*Throwing away ½ the pixels corresponds to sampling the image at a lower frequency. Since you are sampling at a lower frequency, aliasing may result. For example, if the input image was a checkerboard pattern, its pixel values will change very rapidly. Removing ½ the pixels would result in an all-white or all-black image.*

*The most noticeable artifacts will occur if the input image contains high frequencies. If the input image contains frequencies greater than ½ the sampling rate of the output image, the will appear as aliases. Very low frequency images will not have as many aliasing artifacts.*

Develop an improvement to Algorithm 1. Again, show pseudocode and give the mathematical details. Describe why your algorithm is better than just choosing the even pixels.

*A simple way to improve the algorithm is to average blocks of 2x2 pixels. Averaging acts as a box-filter, or low-pass filter, and will attenuate frequencies higher than ½ the output-image sampling rate. Prefiltering an image in this way will antialias the result.*

```cpp
minify( STImage input, STImage output ) {
    for( int y=0; y < output.height; y++ ) {
        int iy = 2*y;
        int jy = 2*y+1
        for( int x = 0; x < output.width; x++ ) {
            int ix = 2*x;
            int jx = 2*x+1
            float v00 = input.GetPixel(ix,iy);
            float v01 = input.GetPixel(ix,jy);
            float v10 = input.GetPixel(jx,iy);
            float v11 = input.GetPixel(jx,jy);
            v = (v00+v01+v10+v11)/4.0;
            output.SetPixel(x,y,v);
        }
    }
}
```
3. Cameras and GLSL (20 points)

We have seen in class how real cameras behave compared to the pinhole camera used in OpenGL. One of the effects that occur when using a real camera is **depth of field**. In this question, you will implement a method that simulates depth of field with GLSL shaders.

Depth of field is an effect caused by the physical nature of lenses. In order for light from a point in the scene to converge to a single point on the film, the source point needs to be a certain distance away from the lens. The plane of point sources that converge to a single point on the **film plane** is called the **plane in focus**. When light rays hit the lens from a source out of focus, the rays don’t converge to one point; instead, they project to a circle on the film. This circle is called the **circle of confusion**. We can simulate this effect by blurring (or averaging) the colors within this circle of confusion for each pixel on screen.

![Diagram of camera and lens with plane in focus and circle of confusion](image)

- **a**: aperture diameter
- **p**: distance to the plane in focus
- **z**: distance to the object you are drawing
- **c**: diameter of the circle of confusion

Thanks to the **thin lens formula**, we know the following:

\[
\frac{1}{z} + \frac{1}{z'} = \frac{1}{f} \\
\frac{1}{p} + \frac{1}{p'} = \frac{1}{f}
\]

Where **f** is the **focal length** (fixed property) of the lens.
A) [8 points] Using the above **diagram** and the **thin lens formula**, derive the expression for \( c \) (diameter of the circle of confusion) as a function of \( f \) (focal length), \( a \) (aperture diameter), \( p \) (plane in focus), \( z \) (depth of the object). Hint: Remember to use similar triangles.

\[
1/z' = 1/f - 1/z \quad \Rightarrow z' = 1 / (1/f - 1/z). \quad \text{Likewise, } p' = 1 / (1/f - 1/p).
\]

*From the diagram, we can see 2 similar triangles:*

- The triangle composed of a lens endpoint, the origin and the point at \( z' \) to the right of the lens
- The triangle composed of one of the endpoints of the circle of confusion on the film plane, the center of the circle of confusion and the point at \( z' \) to the right of the lens.

Thus we have \( \frac{a}{2} / \frac{c}{2} = z' / |z' - p'| \)

\[
\Rightarrow \quad \frac{a}{c} = \frac{z'}{|z' - p'|}
\]

\[
\Rightarrow \quad \frac{c}{a} = |z' - p'| / z'
\]

**Note** that the absolute value around \( z' - p' \) is necessary because the point we’re looking at could also be on the other side of the plane of focus (i.e., \( z \) could be less than \( p \)).

Plug in our expressions for \( z' \) and \( p' \) derived from the thin lens formula, and we get

\[
\Rightarrow \quad c = a * |1 - p'/z'| \quad \text{(refactoring for simplicity)}
\]

\[
\Rightarrow \quad c = a * |1 - (1/f - 1/z) / (1/f - 1/p)|
\]

*There are many ways to factor this equation, and many students chose different ways to express the final result. As long as the student’s result was equivalent to this solution, we gave the student full credit.*

**Rubric:**

- 4 points for correctly formulating the “similar triangles” equation
- 2 points for refactoring the thin lens formulas to isolate the \( z' \) and \( p' \) variables
- 2 points for substituting expressions of \( p' \) and \( z' \) into the equation derived from similar triangles
- -1pt if student forgot the absolute value around the \( p' - z' \) difference term.
- -1pt for minor math mistakes

B) [8 points] Now let’s write some shader code to apply a depth-of-field effect to an OpenGL model. The method we will use is a 2-pass algorithm that uses the depth information from the depth buffer to blur objects depending on their distance to the camera.

Since we cannot access neighboring fragments in GLSL, we need to compute the blur as a post process. We can actually do this in GLSL by copying the output of a shader program from the
frame buffer back into a texture, binding a new shader program, and drawing this texture onto a plane (invoking the new shader program). We can then do arbitrary lookups into this texture to access any of the fragments we drew in our first pass!

A summary of our 2-pass method is as follows:
- **First pass:** write the depth information into a texture, copy the color buffer into a texture. You will assume that this pass has been already done and you will not need to implement it.
- **Second pass:** read the depth and color textures, compute the circle of confusion and apply a blur using this radius.

B1. [5/8 points] Write the `blur()` function that performs a box filter on an image at a given position and within a given radius.

**Notes:** The box filter should compute the average of all the colors contained in a square of size \((2\ast\text{radius}+1)\) and centered at the given position. You only need to output the r,g,b components. Also, to access a neighboring pixel, you need to compute its texel coordinates \((\text{in the } [0,1]\text{ range})\). Note this is an approximation of the circle of confusion blur (it is easier to average over a rectangle than a disk).

**Inputs:** `pixelSize` contains the width and height of a pixel in texture coordinates. The `position` vector is also in texture coordinates.

```glsl
uniform vec2 pixelSize;
vec3 blur (sampler2D image, vec2 position, float radius)
{
    vec3 outputColor = vec3(0.0, 0.0, 0.0);
    float pixwidth = pixelSize.x;
    float pixheight = pixelSize.y;
    float pixWidth = 2.0 * radius + 1.0;
    for (float i = -radius; i <= radius; i++)
        for (float j = -radius; j <= radius; j++)
            outputColor += texture2D(image, position + vec2(i, j) * pixelSize).xyz;
    outputColor *= 1.0 / pow(2.0 * radius + 1.0, 2.0);
    return outputColor;
}
```

- not using `pixelSize`, or not correctly calculating the lookup texel in texel coordinates: -2pt
- not covering the entire blur range of \((2r+1) \times (2r+1)\), -2pt.
- not averaging each contributing pixel in the blur correctly, -2pt.
- For 2 of the above mistakes, we took off -3pts. For all 3, we took off 5 pts.
- Other, more minor GLSL mistakes: -1pt.
B2. [3/8 points] Write the main() function of the 2nd shader that outputs the final color. Here is a description of the inputs and the functions that you should use:

1. the \texttt{coc()} function returns the diameter of the circle of confusion using the formula you derived in part A. \textbf{You don’t need to write this function.}
2. the \texttt{blur()} function returns the average color around the pixel position, within a given radius (in the given texture). You already wrote this function.
3. \texttt{pos} is the texture coordinates of the current fragment in the image, you will use it to make texture look-ups into \texttt{depthTex} and \texttt{imageTex}.
4. \texttt{depthTex} is a greyscale image containing the depth (between 0 and 1) of each fragment visible on screen.
5. \texttt{imageTex} is a copy of the frame buffer from the first shader pass.

```cpp
varying vec2 pos;

uniform sampler2D depthTex;
uniform sampler2D imageTex;

uniform float p;
uniform float f;
uniform float a;

float coc(float z, float p, float f, float a);

void main()
{
    float depth = texture2D(depthTex, pos).x;
    float radius = coc(depth, p, f, a) / 2.0;
    vec3 finalColor = blur(imageTex, pos, radius);

    gl_FragColor = vec4(finalColor,1.);
}
```

-1pt if using diameter as a radius
-1pt for not swizzling out a component of the texture2D sample for the 1D depth.
1 out of 3 points was awarded for a correct “essay response”
C) [2 points] Assume you have no transparent objects in your scene, and you now want to write your color and depth information into a single texture (instead of a color buffer and a separate depth buffer). How would you change the first pass? (Explain in one sentence)

Since all of our objects are completely opaque, we know that all of their alpha components are 1. Thus, we can write the depth of each pixel to the 4\textsuperscript{th} (alpha) channel of the color buffer.

D) [2 points] Describe one limitation of this simple method for simulating depth of field, and some possible artifacts that might result from it. How would you improve the algorithm to deal with this limitation?

There are a few options here.

- The blur kernel is not circular, which leads to some strange aliasing artifacts. We can fix this by using a kernel that more closely approximates a circle (acceptable answer) or by using a Gaussian blur (best answer.)
- Blur function assumes that the depth of objects surrounding the sampled point are the same, which causes strange artifacts when two objects appear abutted in an image, but are actually far apart in the z-direction. (Students chose several ways to illustrate this.) One improvement would be to iterate over every pixel in the source image and compute that pixel’s light contribution to each fragment based on its depth and color information. A less computationally expensive improvement might simply weigh each image texel’s contribution to the fragment by that texel’s depth instead of treating each texel equally.
4. Ray tracing (20 points)

A big advantage of ray tracing is that we aren’t limited to a small set of primitive types like in OpenGL; as long as the shape class provides a method to intersect a ray with the shape, and a method to compute the normal, the shape can be rendered. One way to form more complex geometries is to apply set operators, such as union and difference, to simple primitive shapes.

The union of two shapes                                                                 The difference of two shapes

For this problem, you will write ray intersection routines for the union and difference of two primitive shapes. We will assume the shapes being combined are convex (although the final combined shape may not be convex), so if a ray intersects the shape it will have exactly one entrance point and one exit point.

The following declarations will be useful for this problem:

```c++
//parametric ray struct
struct Ray {
    Point origin;
    Vector dir;
};

/*
myshape.Intersect(r, tnear, nnear, tfar, nfar) will return true if Ray r intersects Shape myshape.
tnear and nnear are the t value and normal where the ray enters the shape, and tfar and nfar are the t and normal for the exit. The normals are oriented to point outwards.
*/
bool Shape::Intersect(Ray r, float &tnear, Normal &nnear, float &tfar, Normal &nfar);
```

Note: Don’t worry about a minimum t and maximum t for the ray for this problem. Just return the first intersection (even if it is behind the ray origin).

A) [8 points] As a warm up, implement ray intersection on the union of two shapes. If there’s an intersection return true and update pint and nint with the point and correctly oriented normal of intersection.

```c++
bool intersect_union( Ray r, Shape a, Shape b, Point &pint, Normal &nint )
```
float tint = INFINITY, tnear, tfar;
Normal nnear, nfar;

if(a.intersect(r, tnear, nnear, tfar, nfar)){
  // tfar always the second intersection so we can ignore
  tint = tnear;
  nint = nnear;
}
if(b.intersect(r, tnear, nnear, tfar, nfar) && tnear < tint){
  tint = tnear;
  nint = nnear;
}

// if tint still equals INF, no intersection
if(tint == INFINITY) return false;
// get point of intersection from ray and t value
pint = r.origin + tint * r.dir;
return true;

• +1 for no intersection case
• +2 for handling case when only A or B is hit
• +2 for handling the case when both are hit (by comparing t values)

• +2 for correct ray to point expression
• +1 for using tnear and nnear from the shape intersection routines (and ignoring tfar/nfar)
B) [12 points] Implement ray intersection for the difference of two shapes, where shape b is subtracted from shape a.

```c
bool intersect_difference( Ray r, Shape a, Shape b, Point &pint, Normal &nint ) {  
  float tnearA, tfarA, tnearB, tfarB;  
  Normal nnearA, nfarA, nnearB, nfarB;  
  
  if(!a.intersect(r, tnearA, nnearA, tfarA, nfarA)) return false;  
  float tint = tnearA;  
  nint = nnearA;  
  if(b.intersect(r, tnearB, nnearB, tfarB, nfarB)){  
    if(tnearB <= tnearA && tnearA < tfarB){  
      if(tfarA <= tfarB) return false; // all of A is cutout  
      // ray has hit the cutout part of the shape  
      tint = tfarB;  
      // flip the normal so it is pointing outward on new shape  
      nint = -1 * nfarB;  
    }  
  }  
  
  pint = r.origin + tint * r.dir;  
  return true;  
}
```

4 points basic cases:
- +2 for returning false if A is not hit
- +2 for returning the first A intersection if B is not hit

8 points for handling the ray hitting both A and B:
There are 4 different intersection orders that need to be handled when both shapes intersect the ray.
- +2 for returning first A intersection if A hit before B
- +2 for handling case where the cutout part of the shape is hit (the second B intersection)
- +2 for flipping the normal of B in this case (also acceptable to orient the normal so it always points in the opposite direction of the ray direction)
- +1 for returning no intersection if the B intersections encompass the A intersections (all of A is cutout along the ray)
- +1 for returning the nearest A intersection if the ray exits shape B before intersecting A

The last two cases were the trickiest, and not heavily weighted. These cases do come up surprisingly often, both would be needed to generate the image in the problem description.
5. Geometric modeling (20 points)

In class, we talked about methods such as Loop Subdivision for representing smooth surfaces. An even simpler way to represent a smooth surface is to construct a surface out of curves. Bezier curves, which we’ve seen in CS 148, are commonly used for this purpose.

A) [8 points] Suppose that you’re given a C++ class declaration for cubic Bezier curves:

class BezierCurve
{
    public:
        Point EvaluateAt(float t);
        Point controlPoints[4];
    }

Provide an implementation of the EvaluateAt method. You can assume that Point has all the standard vector operators defined on it.

There are several correct implementations. Writing out the analytical formula for a Bezier curve is one simple solution:

    return pow(1-t,3) *controlPoints[0] + 3*pow(1-t,2)*t*controlPoints[1]
    + 3*(1-t)*pow(t,2)*controlPoints[2] + pow(t,3)*controlpoints[3];

It’s also possible to arrive at this same expression using Bernstein polynomials:

    int n = 3;
    Point p;
    for (int i = 0; i <= n; i++)
    {
        p += (fact(n) /fact(i)*fact(n-i))*pow(t,i)*pow(1-t,n-i) * 
            controlPoints[i];
    } 
    return p;

The corner-cutting algorithm is also a perfectly good method:

    Point A = (1-t)*controlPoints[0] + t*controlPoints[1];
    Point B = (1-t)*controlPoints[1] + t*controlPoints[2];
    Point C = (1-t)*controlPoints[2] + t*controlPoints[3];
    Point AB = (1-t)*A + t*B;
    Point BC = (1-t)*B + t*C;
    return (1-t)*AB + t*BC;

Some students also used recursive subdivision to evaluate the curve. This works (and was given full credit if implemented correctly), but is more complicated than is absolutely necessary. The recursive subdivision algorithm is intended primarily for drawing curves, rather than evaluating them at a single point.
Scoring rubric:
0 pts – No attempt made, or nonsensical
2 pts – Attempt to use some form of repeated linear interpolation
4 pts – Described correct method, but gave handwavy or incomplete code
6-7 pts – Correct, but made minor math goofs
8 pts – Perfect
B) [12 points] Just as four control points form a cubic Bezier curve, a 4x4 grid of control points forms a *bicubic Bezier surface*. Here’s an illustration of the idea:

Bezier curves are parameterized by a single number “t,” but Bezier surfaces are parameterized by two numbers “u” and “v.” In fact, taking a slice through the surface at a fixed value of “u” or “v” gives you a Bezier curve! Here’s an illustration:

Suppose that you’re given a C++ class declaration for cubic Bezier surfaces:
class BezierSurface
{
    public:
        Point EvaluateAt(float u, float v);
        Point controlPoints[4][4];
}

Provide an implementation of the EvaluateAt method. *Hint: the BezierCurve class from part (a) and its EvaluateAt method will be very useful.*

The general idea here was to use the provided control points to construct 4 curves along the u direction, and to evaluate them each using the provided u value. This gives 4 points which can be connected to form a curve in the v direction. Evaluate this curve using the provided v value to get the final point on the surface.

BezierCurve uCurves[4];
BezierCurve vCurve;
for (int i = 0; i < 4; i++)
{
    uCurves[i].controlPoints = controlPoints[i];
    vCurve.controlPoints[i] = uCurves[i].EvaluateAt(u);
}
return vCurve.EvaluateAt(v);

*Going in the other direction (v first, then u) is also fine.*
*We accepted either order of indexing the control points (controlPoints[u][v] or controlPoints[v][u]).*

Scoring rubric:
0 pts – No attempt made, or nonsensical
3 pts – Made an attempt that had some sense behind it
6 pts – Used interpolation of multiple curves, but incorrectly
9-10 pts – Minor errors in an otherwise correct solution
12 pts – Perfect