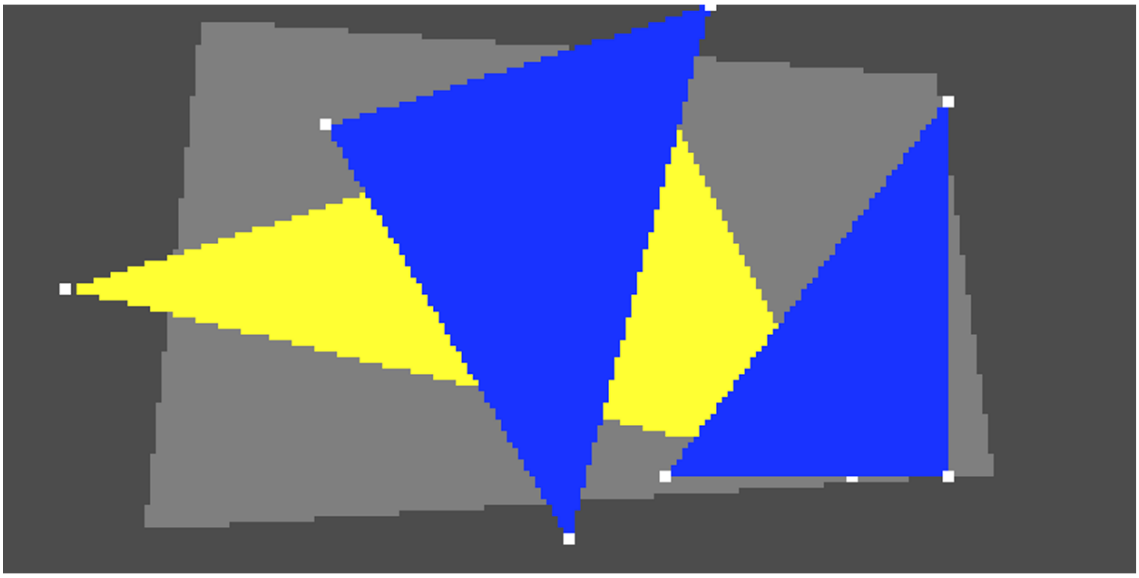
CG Coursework2指导

L.C. 2024.11

实验4对性能要求比较高，实验5不用追求一模一样（事实上，不太可能）。有部分最终代码应该没有在此文件更新修改，敬请谅解。本人分数95+，代码仅供参考。

实验要求

1. 光栅化 (Rasterization)（10分）



- 要求：

  需要完成点是否在多边形内部的判断测试

  已经提供了`isPointInPolygon`函数框架

  需要实现和使用`edge`函数来判断点是否在边的内部或外部

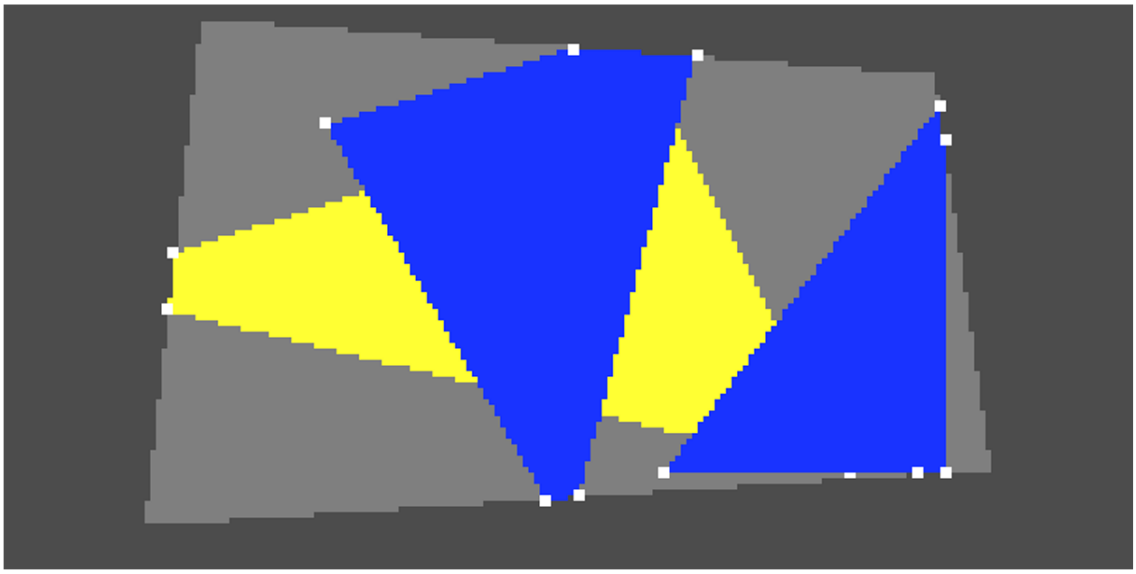
- 主要任务：
  a. 完成`isPointInPolygon`函数的内部循环,实现`edge`函数来判断点与多边形边的关系（10分）

```
// Experiment 1
// Edge function: Determines if a point is on the inner or outer side of an edge
int edge(vec2 point, Vertex a, Vertex b) {
#ifdef SOLUTION_RASTERIZATION
   // TODO: Implement edge function for point-in-polygon test
    // Calculate vectors
    vec2 edgeVector = b.position.xy - a.position.xy;  // Vector from a to b
    vec2 pointVector = point - a.position.xy;         // Vector from a to point

    // Calculate cross product (in 2D)
    // positive: point is on left side, negative: point is on right side
    float crossProduct = edgeVector.x * pointVector.y - edgeVector.y * pointVector.x;
```

```
    // For clockwise winding, points on the right (negative cross
product) are inside
    return (crossProduct <= 0.0) ? INNER_SIDE : OUTER_SIDE;
#endif
    return OUTER_SIDE;
}

// Returns if a point is inside a polygon or not
bool isPointInPolygon(vec2 point, Polygon polygon) {
    // Don't evaluate empty polygons
    if (polygon.vertexCount == 0) return false;
    // Check against each edge of the polygon
    bool rasterise = true;
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i < polygon.vertexCount) {
#ifdef SOLUTION_RASTERIZATION
            // TODO: Complete the inner loop for polygon test
            // Get current vertex and next vertex (wrapping around to
first vertex)
            Vertex currentVertex = polygon.vertices[i];
            Vertex nextVertex = getWrappedPolygonVertex(polygon, i + 1);

            // Test if point is on inner side of current edge
            if (edge(point, currentVertex, nextVertex) == OUTER_SIDE) {
                rasterise = false;
                break;
            }
#else
            rasterise = false;
#endif
        }
    }
    return rasterise;
}
```

- 解决方法:
  - 使用向量叉积来判断点在边的哪一侧
  - 确保对所有边的判断都保持一致的顺时针方向
  - 只有当点在所有边的内侧时才返回true
- 细节指导:
  - `edge`函数实现时需要注意:
    1. 计算向量:(point - a)和(b - a)
    2. 使用叉积判断方向
    3. 根据叉积结果返回INNER_SIDE或OUTER_SIDE
  - `isPointInPolygon`函数实现:
    1. 遍历多边形的每条边
    2. 对每条边调用`edge`函数
    3. 所有边都返回INNER_SIDE才判定点在内部

2. 裁剪 (Clipping)（30分）



- 要求：

  实现Sutherland-Hodgman裁剪算法

  正确处理各种裁剪情况

  确保颜色插值的正确性

- 主要任务：

  a. 实现 `getCrossType` 函数,判断线段与裁剪窗口的交叉类型（10分）

```
// 2a.
// Determines how a polygon edge intersects with a clipping window edge
int getCrossType(Vertex poli1, Vertex poli2, Vertex wind1, Vertex wind2)
{
#ifdef SOLUTION_CLIPPING
    // TODO: Determine intersection type between edges
    // Calculate if endpoints are inside clipping edge using 2D cross
product
    vec2 windowEdge = wind2.position.xy - wind1.position.xy;
    vec2 toPoint1 = poli1.position.xy - wind1.position.xy;
    vec2 toPoint2 = poli2.position.xy - wind1.position.xy;

    float cross1 = windowEdge.x * toPoint1.y - windowEdge.y *
toPoint1.x;
    float cross2 = windowEdge.x * toPoint2.y - windowEdge.y *
toPoint2.x;

    // Determine crossing type based on point positions
    if(cross1 <= 0.0 && cross2 <= 0.0) return INSIDE;      // Both
points inside
    if(cross1 > 0.0 && cross2 > 0.0) return OUTSIDE;       // Both
points outside
    if(cross1 <= 0.0 && cross2 > 0.0) return LEAVING;      // Going from
inside to outside
    return ENTERING;
#else
    return INSIDE;
#endif
}
```

b. 实现 `intersect2D` 函数,计算两条线段的交点（10分）

```
// 2b.
// Calculates the intersection point between two line segments
// This function assumes that the segments are not parallel or
collinear.
Vertex intersect2D(Vertex a, Vertex b, Vertex c, Vertex d) {
#ifdef SOLUTION_CLIPPING
    // TODO: Calculate intersection point of two edges
    // Calculate intersection using parametric form
    vec2 r = b.position.xy - a.position.xy;  // Direction vector of
first line
    vec2 s = d.position.xy - c.position.xy;  // Direction vector of
second line
    vec2 qp = c.position.xy - a.position.xy;

    // Calculate cross products
    float rxs = r.x * s.y - r.y * s.x;
    float qpxr = qp.x * r.y - qp.y * r.x;

    // Calculate intersection parameter t
    float t = (qp.x * s.y - qp.y * s.x) / rxs;

    // Create interpolated vertex
    Vertex result = a;

    // Interpolate position
    result.position = mix(a.position, b.position, t);

    // Perspective-correct color interpolation
    float wa = 1.0 / a.position.w;
    float wb = 1.0 / b.position.w;
    float w = 1.0 / result.position.w;

    // Interpolate color with perspective correction
    result.color = (a.color * wa * (1.0 - t) + b.color * wb * t) /
                   (wa * (1.0 - t) + wb * t);

    // Interpolate texture coordinates
    result.texCoord = mix(a.texCoord, b.texCoord, t);

    return result;
#else
    return a;
#endif
}
```

c. 正确处理顶点颜色的插值,需考虑透视校正（10分）

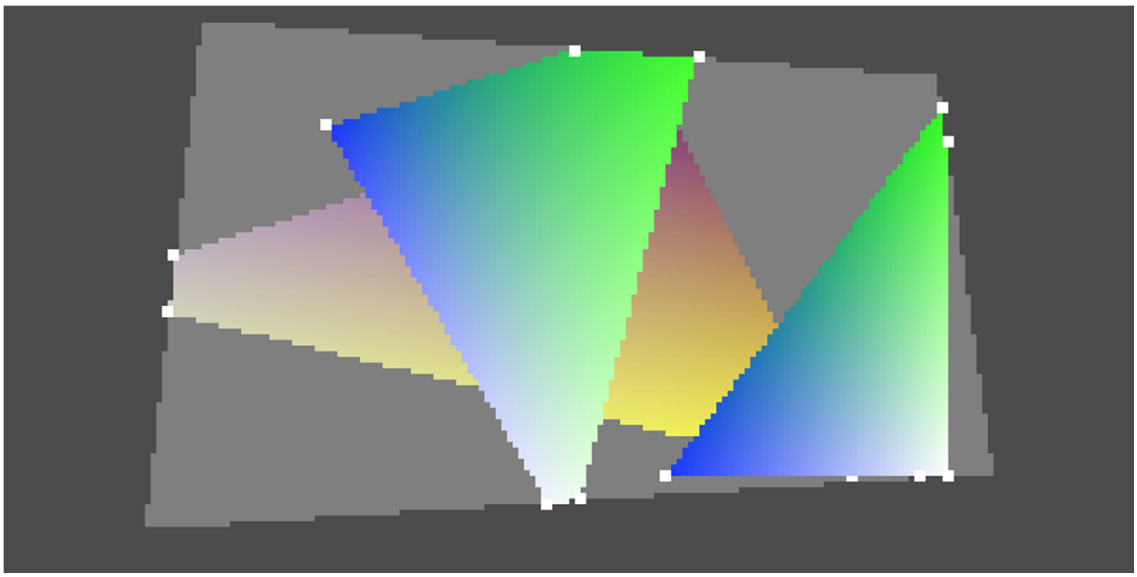
```
//2c.
void sutherlandHodgmanClip(Polygon unclipped, Polygon clipWindow, out
Polygon result) {
    Polygon clipped;
    copyPolygon(clipped, unclipped);
```

```
    // Loop over the clip window
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i >= clipWindow.vertexCount) break;

        // Make a temporary copy of the current clipped polygon
        Polygon oldClipped;
        copyPolygon(oldClipped, clipped);

        // Set the clipped polygon to be empty
        makeEmptyPolygon(clipped);

        // Loop over the current clipped polygon
        for (int j = 0; j < MAX_VERTEX_COUNT; ++j) {
            if (j >= oldClipped.vertexCount) break;

            // Handle the j-th vertex of the clipped polygon. This
should make use of the function
            // intersect() to be implemented above.
#ifdef SOLUTION_CLIPPING
            // TODO: Implement clipping algorithm
            // Get current clipping window edge
            Vertex windowVertex1 = getWrappedPolygonVertex(clipWindow,
i);
            Vertex windowVertex2 = getWrappedPolygonVertex(clipWindow, i
+ 1);

            // Get current polygon vertex and next vertex
            Vertex currentVertex = getWrappedPolygonVertex(oldClipped,
j);
            Vertex nextVertex = getWrappedPolygonVertex(oldClipped, j +
1);

            // Get crossing type
            int crossType = getCrossType(currentVertex, nextVertex,
windowVertex1, windowVertex2);

            // Handle different crossing cases
            if(crossType == INSIDE) {
                appendVertexToPolygon(clipped, nextVertex);
            }
            else if(crossType == LEAVING) {
                Vertex intersectionPoint = intersect2D(currentVertex,
nextVertex,
                                                       windowVertex1,
windowVertex2);
                appendVertexToPolygon(clipped, intersectionPoint);
            }
            else if(crossType == ENTERING) {
                Vertex intersectionPoint = intersect2D(currentVertex,
nextVertex,
                                                       windowVertex1,
windowVertex2);
                appendVertexToPolygon(clipped, intersectionPoint);
                appendVertexToPolygon(clipped, nextVertex);
            }
#else
```

```
            appendVertexToPolygon(clipped,
getWrappedPolygonVertex(oldClipped, j));
#endif
        }
    }

    // Copy the last version to the output
    copyPolygon(result, clipped);
    clipped.textureType = unclipped.textureType;
}
```

- 解决方法:
  - 使用向量和参数方程计算交点
  - 根据点的相对位置判断交叉类型
  - 使用透视校正的插值方法
- 细节指导:
  - `getCrossType`函数实现:
    1. 计算点与裁剪边的关系
    2. 根据关系返回ENTERING/LEAVING/OUTSIDE/INSIDE
  - `intersect2D`函数实现:
    1. 使用参数方程求解交点
    2. 注意处理特殊情况(平行线等)
  - 颜色插值:
    1. 使用重心坐标
    2. 加入透视校正因子(1/w)

3. 插值 (Interpolation) (20分)



- 要求:

  使用重心坐标进行插值

  插值三维坐标和颜色信息

  实现平滑的颜色过渡

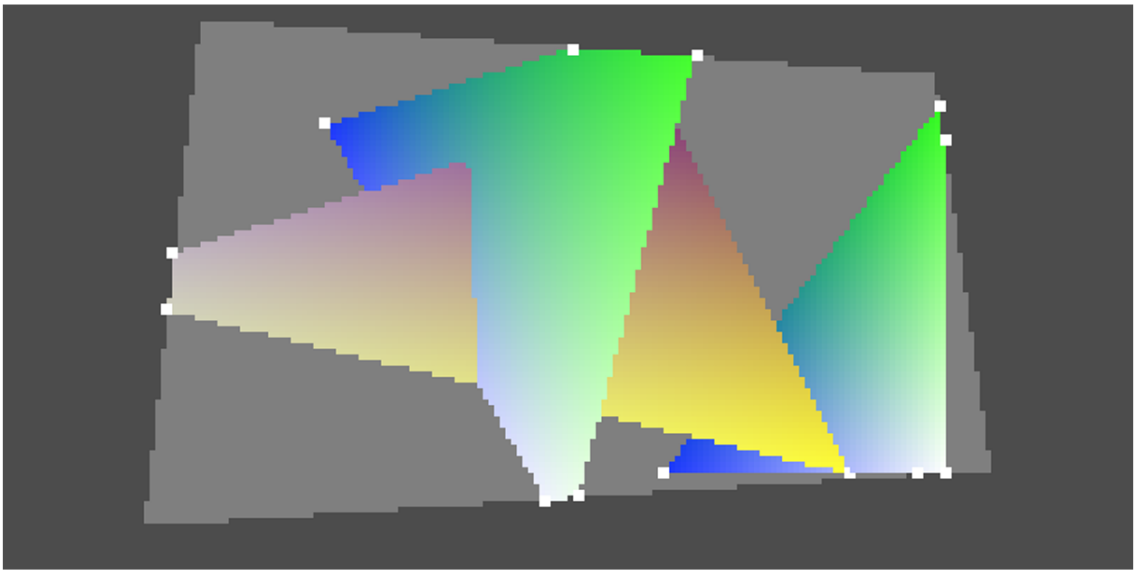
- 主要任务：
  a. 完成 `interpolateVertex` 函数,实现基于重心坐标的顶点插值（20分）

```
// Experiment 3
#if defined(SOLUTION_INTERPOLATION) || defined(SOLUTION_ZBUFFERING)
            // TODO: Implement barycentric interpolation
            // Get three consecutive vertices to form a triangle
    Vertex v0 = polygon.vertices[i];
    Vertex v1 = getWrappedPolygonVertex(polygon, i + 1);
    Vertex v2 = getWrappedPolygonVertex(polygon, i + 2);

    // Calculate total triangle area
    float area = triangleArea(v0.position.xy, v1.position.xy,
v2.position.xy);

    // Calculate barycentric coordinates using areas
    float w0 = triangleArea(point, v1.position.xy, v2.position.xy) /
area;
    float w1 = triangleArea(point, v2.position.xy, v0.position.xy) /
area;
    float w2 = triangleArea(point, v0.position.xy, v1.position.xy) /
area;

    // Perspective correction weights
    float perspWeight0 = w0 / v0.position.w;
    float perspWeight1 = w1 / v1.position.w;
    float perspWeight2 = w2 / v2.position.w;

    weight_corr_sum += perspWeight0 + perspWeight1 + perspWeight2;
    weight_sum += w0 + w1 + w2;

    // Accumulate all attributes with proper weights
    colorSum += v0.color * perspWeight0 + v1.color * perspWeight1 +
v2.color * perspWeight2;
    texCoordSum += v0.texCoord * perspWeight0 + v1.texCoord *
perspWeight1 + v2.texCoord * perspWeight2;
    positionSum += v0.position * perspWeight0 + v1.position *
perspWeight1 + v2.position * perspWeight2;

#endif

// Experiment 4
#ifdef SOLUTION_ZBUFFERING
            // TODO
            // Accumulate position with perspective correction
#endif

// Experiment 3
#ifdef SOLUTION_INTERPOLATION
            // TODO
            // Accumulate color with perspective correction
#endif

// Experiment 6
#ifdef SOLUTION_TEXTURING
```

```
            // TODO
            // Accumulate texture coordinates with perspective
correction
#endif
        }
    }
    Vertex result = polygon.vertices[0];

// Experiment 3
#ifdef SOLUTION_INTERPOLATION
    // TODO
    // Normalize the interpolated colors and texture coordinates
    result.color = colorSum / weight_corr_sum;
    result.texCoord = texCoordSum / weight_corr_sum;
#endif

// Experiment 4
#ifdef SOLUTION_ZBUFFERING
    // TODO
    // Normalize the interpolated position
    result.position = positionSum / weight_sum;
#endif

// Experiment 6
#ifdef SOLUTION_TEXTURING
    // TODO
    // Normalize the interpolated texture coordinates
    result.texCoord = texCoordSum / weight_corr_sum;
#endif

  return result;
}
```

- 解决方法:
  - 计算重心坐标
  - 使用重心坐标进行加权插值
  - 处理透视校正
- 细节指导:
  - 重心坐标计算:
    1. 计算各子三角形面积
    2. 归一化得到权重
  - 插值实现:
    1. 对位置和颜色分别进行加权
    2. 注意深度值的插值
    3. 确保权重和为1

4. Z缓冲 (Z-buffering)（10分）



- 要求：

  在光栅化代码中增加深度测试

  确保正确处理重叠三角形的可见性

  位置需要正确插值以进行深度测试

- 主要任务：

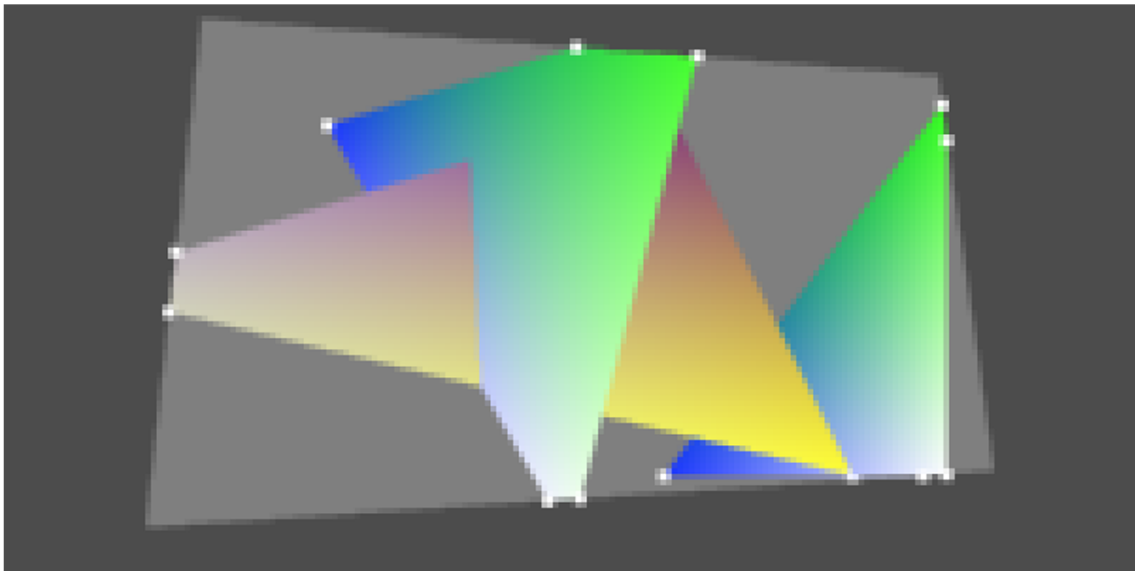
  a. 在 `drawPolygon` 函数中实现Z缓冲算法,处理多边形的遮挡关系（10分）

```
// Experiment 4
void drawPolygon(
      vec2 point,
  Polygon clipWindow,
  Polygon oldPolygon,
  inout vec3 color,
  inout float depth)
{
    Polygon projectedPolygon;
    projectPolygon(projectedPolygon, oldPolygon);

    Polygon clippedPolygon;
    sutherlandHodgmanClip(projectedPolygon, clipWindow, clippedPolygon);

    if (isPointInPolygon(point, clippedPolygon)) {

        Vertex interpolatedVertex =
          interpolateVertex(point, projectedPolygon);
#ifdef SOLUTION_ZBUFFERING
        // TODO: Implement depth testing
        // Perform depth test
        float newDepth = interpolatedVertex.position.z;
        if (newDepth < depth) {
            // Update color and depth if the new point is closer
            color = getInterpVertexColor(interpolatedVertex,
oldPolygon.textureType);
            depth = newDepth;
        }
#else
```

```
        color = getInterpVertexColor(interpolatedVertex,
oldPolygon.textureType);
        depth = interpolatedVertex.position.z;
#endif
    }

    if (isPointOnPolygonVertex(point, clippedPolygon)) {
         color = vec3(1);
    }
}
```

- 解决方法:
  - 维护深度缓冲区
  - 在每个像素进行深度比较
  - 只更新深度值更小的片段
- 细节指导:
  - Z缓冲实现步骤:
    1. 在interpolateVertex中正确插值z值
    2. 比较当前片段的深度值与已存储的深度值
    3. 只有当新的深度值更小时才更新颜色和深度
    4. 注意深度值的透视校正

5. 抗锯齿 (Anti-aliasing)（15分）



- 要求:

  实现高质量的抗锯齿效果

  不能仅仅是模糊处理

  需要提供质量和性能的平衡选项

- 主要任务:

  a. 实现抗锯齿算法（13分）

```
// Experiment 5
void main() {
    vec3 color = vec3(0);
```

```
#ifdef SOLUTION_AALIAS
    // Define constants for quality control
    #define FAST_MODE  // Comment this line to use high quality mode

    #ifdef FAST_MODE
        // Fast mode: 2x MSAA implementation
        const int numSamples = 2;
        vec2 sampleOffsets[numSamples];
        // Standard 2x MSAA sample pattern
        // Rotated grid pattern for better coverage
        sampleOffsets[0] = vec2(-0.25,  0.25);   // Upper left
        sampleOffsets[1] = vec2(0.25, -0.25);  // Lower rightcenter

    #else

        // High quality mode: 8x MSAA implementation
        const int numSamples = 8;
        vec2 sampleOffsets[numSamples];
        // Standard 8x MSAA sample pattern
        // Using rotated grid pattern (similar to DX11 standard pattern)
        sampleOffsets[0]  = vec2( 0.0625, -0.1875);  // Sample 1
        sampleOffsets[1]  = vec2(-0.0625,  0.1875);  // Sample 2
        sampleOffsets[2]  = vec2( 0.3125,  0.0625);  // Sample 3
        sampleOffsets[3]  = vec2(-0.1875, -0.3125);  // Sample 4
        sampleOffsets[4]  = vec2(-0.3125,  0.3125);  // Sample 5
        sampleOffsets[5]  = vec2( 0.1875,  0.3125);  // Sample 6
        sampleOffsets[6]  = vec2( 0.3125, -0.1875);  // Sample 7
        sampleOffsets[7]  = vec2(-0.3125, -0.0625);  // Sample 8

    #endif

    // Accumulate color from samples
    vec3 accumColor = vec3(0.0);

    // Sample multiple points around the current pixel
    for (int i = 0; i < numSamples; i++) {
        vec2 samplePos = gl_FragCoord.xy + sampleOffsets[i];

        // Get color for this sample
        vec3 sampleColor;
        drawScene(samplePos, sampleColor);
        accumColor += sampleColor;
    }

    // Average the accumulated samples
    color = accumColor / float(numSamples);

#else
    drawScene(gl_FragCoord.xy, color);
#endif

    gl_FragColor.rgb = color;
    gl_FragColor.a = 1.0;
}
```

b. 提供质量和速度的权衡选项（2分）

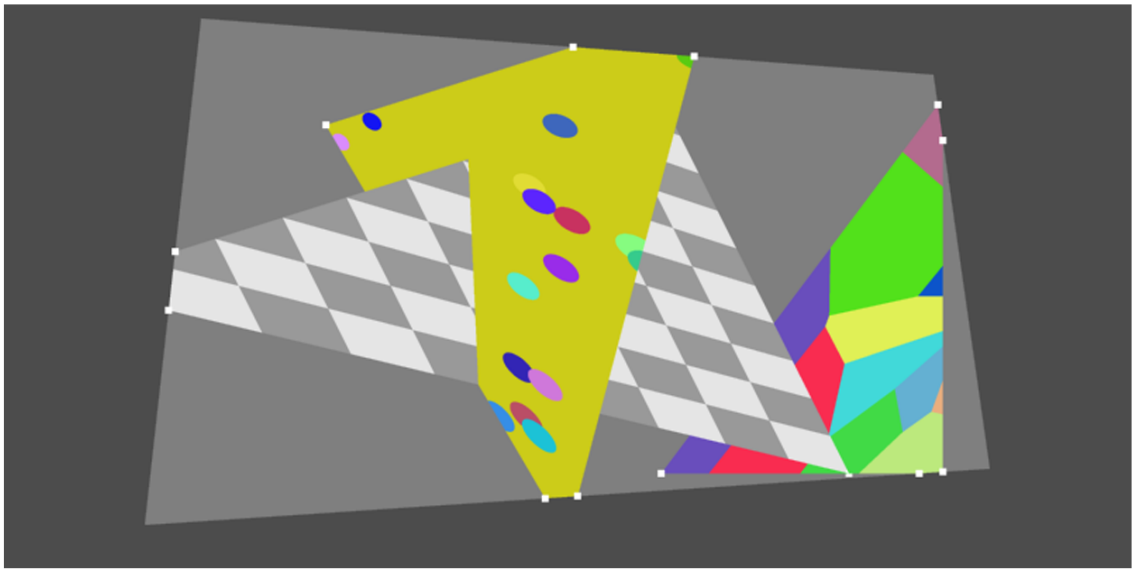
```
通过注释来切换：
#define FAST_MODE  // Comment this line to use high quality mode
```

- 解决方法:

- 实现MSAA(多重采样抗锯齿)
  - 使用自适应采样
- 提供可配置的采样数量
- 细节指导:
  - MSAA实现步骤:

```
1. 对每个像素进行多点采样
   2. 根据采样点是否在图形内部计算覆盖率
3. 根据覆盖率进行颜色混合
 - 性能优化：
   1. 提供可配置的采样数量(如4x,8x,16x)
   2. 实现自适应采样(边缘处采样更多)
3. 在shader中添加质量控制参数
```

6. 纹理 (Texturing)（15分）



- 要求：

  实现三种不同的程序化纹理

  正确处理UV坐标的映射

  确保纹理图案的清晰度和连续性

- 主要任务：

  a. 实现棋盘格纹理: 灰白相间,UV空间中64个方格（5分）

```
// Experiment 6
// 6a.
// Checkerboard texture implementation
vec3 textureCheckerboard(vec2 texCoord) {
```

```
#ifdef SOLUTION_TEXTURING
    // Scale UV coordinates to create 8x8 grid (64 squares total)
    vec2 scaledUV = texCoord *10.0;

    // Get integer cell coordinates
    ivec2 cell = ivec2(floor(scaledUV));

    // Determine if we're in a white or gray cell
    bool isWhite = (intModulo(cell.x + cell.y, 2) == 0);

    // Return white or gray color (changed from 0.5 to 0.8 for lighter
gray)
    return isWhite ? vec3(1.0) : vec3(0.65);
#endif
    return vec3(1.0, 0.0, 0.0);
}
```

b. 实现波尔卡圆点纹理: 约20个随机颜色圆点,白色背景（5分）

```
// 6b.
// Polka dot texture implementation
vec3 texturePolkadot(vec2 texCoord)
{
    const vec3 bgColor = vec3(0.8, 0.8, 0.1);
    // This implementation is global, adding a set number of dots at
random to the whole texture.
    prngSeed = globalPrngSeed; // Need to reseed here to play nicely
with anti-aliasing
    const int nPolkaDots = 30;
    const float polkaDotRadius = 0.03;
    vec3 color = bgColor;

    #ifdef SOLUTION_TEXTURING
    // Generate polka dots
    for(int i = 0; i < nPolkaDots; ++i) {
        // Generate random dot position and color
        vec2 dotCenter = vec2(prngUniform01(), prngUniform01());
            if (distance(texCoord, dotCenter) < polkaDotRadius) {
                color = vec3(randomColor());
            }
        }
    #endif
    return color;
}
```

c. 实现细胞纹理: 约20个随机颜色的Voronoi单元格（5分）

```
// 6c.
// Voronoi (cellular) texture implementation
vec3 textureVoronoi(vec2 texCoord)
{
    // This implementation is global, adding a set number of cells at
random to the whole texture.
```

```
    prngSeed = globalPrngSeed; // Need to reseed here to play nicely
with anti-aliasing
    const int nVoronoiCells = 15;

    #ifdef SOLUTION_TEXTURING
    float minDist = 1.0;  // Track the minimum distance to any cell
center
    vec3 cellColor = vec3(0.0, 0.0, 1.0);  // Default color if no cells
are found

    // First, generate positions for all Voronoi cells
    vec2 cellCenters[15];  // Array to store cell center positions

    // Generate random positions for all cells
    for(int i = 0; i < nVoronoiCells; i++) {
        cellCenters[i] = vec2(prngUniform01(), prngUniform01());
    }

    // Generate colors and find nearest cell in a single pass
    for(int i = 0; i < nVoronoiCells; i++) {
        vec3 currentColor = randomColor();  // Generate color for
current cell
        float dist = length(texCoord - cellCenters[i]);
        if(dist < minDist) {
            minDist = dist;
            cellColor = currentColor;  // Use color if this is the
nearest cell
        }
    }

    return cellColor;
#endif
    return vec3(0.0, 0.0, 1.0);  // Default return value if
SOLUTION_TEXTURING is not defined
}
```

```
// Experiment 6
// Color interpolation with texture support
vec3 getInterpVertexColor(Vertex interpVertex, int textureType)
{
    #ifdef SOLUTION_TEXTURING
    // Choose appropriate texture based on type
    if (textureType == TEXTURE_CHECKERBOARD) {
        return textureCheckerboard(interpVertex.texCoord);
    } else if (textureType == TEXTURE_POLKADOT) {
        return texturePolkadot(interpVertex.texCoord);
    } else if (textureType == TEXTURE_VORONOI) {
        return textureVoronoi(interpVertex.texCoord);
    }
    return interpVertex.color;
    #else
    return interpVertex.color;
    #endif
    return vec3(1.0, 0.0, 1.0);
}
```

- 解决方法:
  - 使用UV坐标计算纹理模式
  - 实现随机数生成器
  - 使用数学函数生成几何图案
- 细节指导:
  - 棋盘格纹理:
    1. 使用floor函数将UV坐标分割成网格
    2. 使用模运算确定每个格子的颜色
    3. 保证边界过渡的清晰度
  - 波尔卡圆点:
    1. 生成随机的圆点中心和颜色
    2. 计算像素到圆点中心的距离
    3. 使用smoothstep实现边缘平滑
  - 细胞纹理:
    1. 生成随机的细胞中心点
    2. 计算每个像素到最近中心点的距离
    3. 根据最近的中心点确定颜色

完整代码部分(带注释)

```
#define SOLUTION_RASTERIZATION
#define SOLUTION_CLIPPING
#define SOLUTION_INTERPOLATION
#define SOLUTION_ZBUFFERING
#define SOLUTION_AALIAS
#define SOLUTION_TEXTURING

precision highp float;
uniform float time;

// Polygon / vertex functionality
const int MAX_VERTEX_COUNT = 8;

uniform ivec2 viewport;

struct Vertex {
    vec4 position;
    vec3 color;
    vec2 texCoord;
};

const int TEXTURE_NONE = 0;
const int TEXTURE_CHECKERBOARD = 1;
const int TEXTURE_POLKADOT = 2;
const int TEXTURE_VORONOI = 3;
```

```
const int globalPrngSeed = 7;

struct Polygon {
    // Numbers of vertices, i.e., points in the polygon
    int vertexCount;
    // The vertices themselves
    Vertex vertices[MAX_VERTEX_COUNT];
    int textureType;
};

// Appends a vertex to a polygon
void appendVertexToPolygon(inout Polygon polygon, Vertex element) {
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i == polygon.vertexCount) {
            polygon.vertices[i] = element;
        }
    }
    polygon.vertexCount++;
}

// Copy Polygon source to Polygon destination
void copyPolygon(inout Polygon destination, Polygon source) {
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        destination.vertices[i] = source.vertices[i];
    }
    destination.vertexCount = source.vertexCount;
    destination.textureType = source.textureType;
}

// Get the i-th vertex from a polygon, but when asking for the one behind the
last, get the first again
Vertex getWrappedPolygonVertex(Polygon polygon, int index) {
    if (index >= polygon.vertexCount) index -= polygon.vertexCount;
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i == index) return polygon.vertices[i];
    }
}

// Creates an empty polygon
void makeEmptyPolygon(out Polygon polygon) {
  polygon.vertexCount = 0;
}

// Clipping part

#define ENTERING 0
#define LEAVING 1
#define OUTSIDE 2
#define INSIDE 3

// Experiment 2

// 2a.
// Determines how a polygon edge intersects with a clipping window edge
int getCrossType(Vertex poli1, Vertex poli2, Vertex wind1, Vertex wind2) {
```

```
#ifdef SOLUTION_CLIPPING
    // TODO: Determine intersection type between edges
    // Calculate if endpoints are inside clipping edge using 2D cross product
    vec2 windowEdge = wind2.position.xy - wind1.position.xy;
    vec2 toPoint1 = poli1.position.xy - wind1.position.xy;
    vec2 toPoint2 = poli2.position.xy - wind1.position.xy;

    float cross1 = windowEdge.x * toPoint1.y - windowEdge.y * toPoint1.x;
    float cross2 = windowEdge.x * toPoint2.y - windowEdge.y * toPoint2.x;

    // Determine crossing type based on point positions
    if(cross1 <= 0.0 && cross2 <= 0.0) return INSIDE;      // Both points inside
    if(cross1 > 0.0 && cross2 > 0.0) return OUTSIDE;       // Both points outside
    if(cross1 <= 0.0 && cross2 > 0.0) return LEAVING;      // Going from inside
to outside
    return ENTERING;
#else
    return INSIDE;
#endif
}

// 2b.
// Calculates the intersection point between two line segments
// This function assumes that the segments are not parallel or collinear.
Vertex intersect2D(Vertex a, Vertex b, Vertex c, Vertex d) {
#ifdef SOLUTION_CLIPPING
    // TODO: Calculate intersection point of two edges
    // Calculate intersection using parametric form
    vec2 r = b.position.xy - a.position.xy;  // Direction vector of first line
    vec2 s = d.position.xy - c.position.xy;  // Direction vector of second line
    vec2 qp = c.position.xy - a.position.xy;

    // Calculate cross products
    float rxs = r.x * s.y - r.y * s.x;
    float qpxr = qp.x * r.y - qp.y * r.x;

    // Calculate intersection parameter t
    float t = (qp.x * s.y - qp.y * s.x) / rxs;

    // Create interpolated vertex
    Vertex result = a;

    // Interpolate position
    result.position = mix(a.position, b.position, t);

    // Perspective-correct color interpolation
    float wa = 1.0 / a.position.w;
    float wb = 1.0 / b.position.w;
    float w = 1.0 / result.position.w;

    // Interpolate color with perspective correction
    result.color = (a.color * wa * (1.0 - t) + b.color * wb * t) /
                   (wa * (1.0 - t) + wb * t);

    // Interpolate texture coordinates
    result.texCoord = mix(a.texCoord, b.texCoord, t);
```

```
    return result;
#else
    return a;
#endif
}

//2c.
void sutherlandHodgmanClip(Polygon unclipped, Polygon clipWindow, out Polygon
result) {
    Polygon clipped;
    copyPolygon(clipped, unclipped);

    // Loop over the clip window
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i >= clipWindow.vertexCount) break;

        // Make a temporary copy of the current clipped polygon
        Polygon oldClipped;
        copyPolygon(oldClipped, clipped);

        // Set the clipped polygon to be empty
        makeEmptyPolygon(clipped);

        // Loop over the current clipped polygon
        for (int j = 0; j < MAX_VERTEX_COUNT; ++j) {
            if (j >= oldClipped.vertexCount) break;

            // Handle the j-th vertex of the clipped polygon. This should make
use of the function
            // intersect() to be implemented above.
#ifdef SOLUTION_CLIPPING
            // TODO: Implement clipping algorithm
            // Get current clipping window edge
            Vertex windowVertex1 = getWrappedPolygonVertex(clipWindow, i);
            Vertex windowVertex2 = getWrappedPolygonVertex(clipWindow, i + 1);

            // Get current polygon vertex and next vertex
            Vertex currentVertex = getWrappedPolygonVertex(oldClipped, j);
            Vertex nextVertex = getWrappedPolygonVertex(oldClipped, j + 1);

            // Get crossing type
            int crossType = getCrossType(currentVertex, nextVertex,
windowVertex1, windowVertex2);

            // Handle different crossing cases
            if(crossType == INSIDE) {
                appendVertexToPolygon(clipped, nextVertex);
            }
            else if(crossType == LEAVING) {
                Vertex intersectionPoint = intersect2D(currentVertex, nextVertex,
                                                       windowVertex1,
windowVertex2);
                appendVertexToPolygon(clipped, intersectionPoint);
            }
            else if(crossType == ENTERING) {
```

```
                Vertex intersectionPoint = intersect2D(currentVertex, nextVertex,
                                                       windowVertex1,
windowVertex2);
                appendVertexToPolygon(clipped, intersectionPoint);
                appendVertexToPolygon(clipped, nextVertex);
            }
#else
            appendVertexToPolygon(clipped, getWrappedPolygonVertex(oldClipped,
j));
#endif
        }
    }

    // Copy the last version to the output
    copyPolygon(result, clipped);
    clipped.textureType = unclipped.textureType;
}

// SOLUTION_RASTERIZATION and culling part

#define INNER_SIDE 0
#define OUTER_SIDE 1

// Assuming a clockwise (vertex-wise) polygon, returns whether the input point
// is on the inner or outer side of the edge (ab)


// Experiment 1
// Edge function: Determines if a point is on the inner or outer side of an edge
int edge(vec2 point, Vertex a, Vertex b) {
#ifdef SOLUTION_RASTERIZATION
   // TODO: Implement edge function for point-in-polygon test
    // Calculate vectors
    vec2 edgeVector = b.position.xy - a.position.xy;  // Vector from a to b
    vec2 pointVector = point - a.position.xy;         // Vector from a to point

    // Calculate cross product (in 2D)
    // positive: point is on left side, negative: point is on right side
    float crossProduct = edgeVector.x * pointVector.y - edgeVector.y *
pointVector.x;

    // For clockwise winding, points on the right (negative cross product) are
inside
    return (crossProduct <= 0.0) ? INNER_SIDE : OUTER_SIDE;
#endif
    return OUTER_SIDE;
}

// Returns if a point is inside a polygon or not
bool isPointInPolygon(vec2 point, Polygon polygon) {
    // Don't evaluate empty polygons
    if (polygon.vertexCount == 0) return false;
    // Check against each edge of the polygon
    bool rasterise = true;
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i < polygon.vertexCount) {
```

```
#ifdef SOLUTION_RASTERIZATION
            // TODO: Complete the inner loop for polygon test
            // Get current vertex and next vertex (wrapping around to first
vertex)
            Vertex currentVertex = polygon.vertices[i];
            Vertex nextVertex = getWrappedPolygonVertex(polygon, i + 1);

            // Test if point is on inner side of current edge
            if (edge(point, currentVertex, nextVertex) == OUTER_SIDE) {
                rasterise = false;
                break;
            }
#else
            rasterise = false;
#endif
        }
    }
    return rasterise;
}

bool isPointOnPolygonVertex(vec2 point, Polygon polygon) {
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i < polygon.vertexCount) {
            ivec2 pixelDifference = ivec2(abs(polygon.vertices[i].position.xy -
point) * vec2(viewport));
            int pointSize = viewport.x / 200;
            if( pixelDifference.x <= pointSize && pixelDifference.y <= pointSize)
{
              return true;
            }
        }
    }
    return false;
}

float triangleArea(vec2 a, vec2 b, vec2 c) {
    // https://en.wikipedia.org/wiki/Heron%27s_formula
    float ab = length(a - b);
    float bc = length(b - c);
    float ca = length(c - a);
    float s = (ab + bc + ca) / 2.0;
    return sqrt(max(0.0, s * (s - ab) * (s - bc) * (s - ca)));
}


// Experiment 3/4/6
Vertex interpolateVertex(vec2 point, Polygon polygon) {
    vec3 colorSum = vec3(0.0);
    vec4 positionSum = vec4(0.0);
    vec2 texCoordSum = vec2(0.0);
    float weight_sum = 0.0;
    float weight_corr_sum = 0.0;

    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i < polygon.vertexCount) {
```

```
// Experiment 3
#if defined(SOLUTION_INTERPOLATION) || defined(SOLUTION_ZBUFFERING)
            // TODO: Implement barycentric interpolation
            // Get three consecutive vertices to form a triangle
    Vertex v0 = polygon.vertices[i];
    Vertex v1 = getWrappedPolygonVertex(polygon, i + 1);
    Vertex v2 = getWrappedPolygonVertex(polygon, i + 2);

    // Calculate total triangle area
    float area = triangleArea(v0.position.xy, v1.position.xy, v2.position.xy);

    // Calculate barycentric coordinates using areas
    float w0 = triangleArea(point, v1.position.xy, v2.position.xy) / area;
    float w1 = triangleArea(point, v2.position.xy, v0.position.xy) / area;
    float w2 = triangleArea(point, v0.position.xy, v1.position.xy) / area;

    // Perspective correction weights
    float perspWeight0 = w0 / v0.position.w;
    float perspWeight1 = w1 / v1.position.w;
    float perspWeight2 = w2 / v2.position.w;

    weight_corr_sum += perspWeight0 + perspWeight1 + perspWeight2;
    weight_sum += w0 + w1 + w2;

    // Accumulate all attributes with proper weights
    colorSum += v0.color * perspWeight0 + v1.color * perspWeight1 + v2.color *
perspWeight2;
    texCoordSum += v0.texCoord * perspWeight0 + v1.texCoord * perspWeight1 +
v2.texCoord * perspWeight2;
    positionSum += v0.position * w0 + v1.position * w1 + v2.position * w2;

#endif

// Experiment 4
#ifdef SOLUTION_ZBUFFERING
            // TODO
            // Accumulate position with perspective correction
#endif

// Experiment 3
#ifdef SOLUTION_INTERPOLATION
            // TODO
            // Accumulate color with perspective correction
#endif

// Experiment 6
#ifdef SOLUTION_TEXTURING
            // TODO
            // Accumulate texture coordinates with perspective correction
#endif
        }
    }
    Vertex result = polygon.vertices[0];

// Experiment 3
#ifdef SOLUTION_INTERPOLATION
```

```
    // TODO
    // Normalize the interpolated colors and texture coordinates
    result.color = colorSum / weight_corr_sum;
    result.texCoord = texCoordSum / weight_corr_sum;
#endif

// Experiment 4
#ifdef SOLUTION_ZBUFFERING
    // TODO
    // Normalize the interpolated position
    result.position = positionSum / weight_sum;
#endif

// Experiment 6
#ifdef SOLUTION_TEXTURING
    // TODO
    // Normalize the interpolated texture coordinates
    result.texCoord = texCoordSum / weight_corr_sum;
#endif

  return result;
}

// Projection part

// Used to generate a projection matrix.
mat4 computeProjectionMatrix() {
    mat4 projectionMatrix = mat4(1);

    float aspect = float(viewport.x) / float(viewport.y);
    float imageDistance = 2.0;

    float xMin = -0.5;
    float yMin = -0.5;
    float xMax = 0.5;
    float yMax = 0.5;


    mat4 regPyr = mat4(1.0);
    float d = imageDistance;

    float w = xMax - xMin;
    float h = (yMax - yMin) / aspect;
    float x = xMax + xMin;
    float y = yMax + yMin;

    regPyr[0] = vec4(d / w, 0, 0, 0);
    regPyr[1] = vec4(0, d / h, 0, 0);
    regPyr[2] = vec4(-x/w, -y/h, 1, 0);
    regPyr[3] = vec4(0,0,0,1);

    // Scale by 1/D
    mat4 scaleByD = mat4(1.0/d);
    scaleByD[3][3] = 1.0;

    // Perspective Division
```

```
    mat4 perspDiv = mat4(1.0);
    perspDiv[2][3] = 1.0;

    projectionMatrix = perspDiv * scaleByD * regPyr;


    return projectionMatrix;
}

// Used to generate a simple "look-at" camera.
mat4 computeViewMatrix(vec3 VRP, vec3 TP, vec3 VUV) {
    mat4 viewMatrix = mat4(1);

    // The VPN is pointing away from the TP. Can also be modeled the other way
around.
    vec3 VPN = TP - VRP;

    // Generate the camera axes.
    vec3 n = normalize(VPN);
    vec3 u = normalize(cross(VUV, n));
    vec3 v = normalize(cross(n, u));

    viewMatrix[0] = vec4(u[0], v[0], n[0], 0);
    viewMatrix[1] = vec4(u[1], v[1], n[1], 0);
    viewMatrix[2] = vec4(u[2], v[2], n[2], 0);
    viewMatrix[3] = vec4(-dot(VRP, u), -dot(VRP, v), -dot(VRP, n), 1);
    return viewMatrix;
}

vec3 getCameraPosition() {
    //return 10.0 * vec3(sin(time * 1.3), 0, cos(time * 1.3));
    return 10.0 * vec3(sin(0.0), 0, cos(0.0));
}

// Takes a single input vertex and projects it using the input view and
projection matrices
vec4 projectVertexPosition(vec4 position) {

  // Set the parameters for the look-at camera.
    vec3 TP = vec3(0, 0, 0);
    vec3 VRP = getCameraPosition();
    vec3 VUV = vec3(0, 1, 0);

    // Compute the view matrix.
    mat4 viewMatrix = computeViewMatrix(VRP, TP, VUV);

  // Compute the projection matrix.
    mat4 projectionMatrix = computeProjectionMatrix();

    vec4 projectedVertex = projectionMatrix * viewMatrix * position;
    projectedVertex.xyz = (projectedVertex.xyz / projectedVertex.w);
    return projectedVertex;
}

// Projects all the vertices of a polygon
void projectPolygon(inout Polygon projectedPolygon, Polygon polygon) {
```

```
    copyPolygon(projectedPolygon, polygon);
    for (int i = 0; i < MAX_VERTEX_COUNT; ++i) {
        if (i < polygon.vertexCount) {
            projectedPolygon.vertices[i].position =
projectVertexPosition(polygon.vertices[i].position);
        }
    }
}

int intModulo(int a, int b)
{
    // Manual implementation of mod for int; note the % operator & mod for int
isn't supported in some WebGL versions.
    return a - (a/b)*b;
}

// Experiment 6
// 6a.
// Checkerboard texture implementation
vec3 textureCheckerboard(vec2 texCoord) {
#ifdef SOLUTION_TEXTURING
    // Scale UV coordinates to create 8x8 grid (64 squares total)
    vec2 scaledUV = texCoord *10.0;

    // Get integer cell coordinates
    ivec2 cell = ivec2(floor(scaledUV));

    // Determine if we're in a white or gray cell
    bool isWhite = (intModulo(cell.x + cell.y, 2) == 0);

    // Return white or gray color (changed from 0.5 to 0.8 for lighter gray)
    return isWhite ? vec3(1.0) : vec3(0.65);
#endif
    return vec3(1.0, 0.0, 0.0);
}

int prngSeed = 5;
const int prngMult = 174763; // This is a prime
const float maxUint = 2147483647.0; // Max magnitude of a 32-bit signed integer

float prngUniform01()
{
    // Very basic linear congruential generator
(https://en.wikipedia.org/wiki/Lehmer_random_number_generator)
    // Using signed integers (as some WebGL doesn't support unsigned).
    prngSeed *= prngMult;
    // Now the seed is a "random" value between -2147483648 and 2147483647.
    // Convert to float and scale to the 0,1 range.
    float val = float(prngSeed) / maxUint;
    return 0.5 + (val * 0.5);
}

float prngUniform(float min, float max)
{
    return prngUniform01() * (max - min) + min;
}
```

```
vec3 hsv2rgb(vec3 c)
{
    vec4 K = vec4(1.0, 2.0 / 3.0, 1.0 / 3.0, 3.0);
    vec3 p = abs(fract(c.xxx + K.xyz) * 6.0 - K.www);
    return c.z * mix(K.xxx, clamp(p - K.xxx, 0.0, 1.0), c.y);
}

vec3 randomColor()
{
    return hsv2rgb(vec3(prngUniform01(), prngUniform(0.4, 1.0), prngUniform(0.7,
1.0)));
}

// 6b.
// Polka dot texture implementation
vec3 texturePolkadot(vec2 texCoord)
{
    const vec3 bgColor = vec3(0.8, 0.8, 0.1);
    // This implementation is global, adding a set number of dots at random to
the whole texture.
    prngSeed = globalPrngSeed; // Need to reseed here to play nicely with anti-
aliasing
    const int nPolkaDots = 30;
    const float polkaDotRadius = 0.03;
    vec3 color = bgColor;

    #ifdef SOLUTION_TEXTURING
    // Generate polka dots
    for(int i = 0; i < nPolkaDots; ++i) {
        // Generate random dot position and color
        vec2 dotCenter = vec2(prngUniform01(), prngUniform01());
            if (distance(texCoord, dotCenter) < polkaDotRadius) {
                color = vec3(randomColor());
            }
        }
    #endif
    return color;
}

// 6c.
// Voronoi (cellular) texture implementation
vec3 textureVoronoi(vec2 texCoord)
{
    // This implementation is global, adding a set number of cells at random to
the whole texture.
    prngSeed = globalPrngSeed; // Need to reseed here to play nicely with anti-
aliasing
    const int nVoronoiCells = 15;

    #ifdef SOLUTION_TEXTURING
    float minDist = 1.0;  // Track the minimum distance to any cell center
    vec3 cellColor = vec3(0.0, 0.0, 1.0);  // Default color if no cells are found

    // First, generate positions for all Voronoi cells
    vec2 cellCenters[15];  // Array to store cell center positions
```

```
    // Generate random positions for all cells
    for(int i = 0; i < nVoronoiCells; i++) {
        cellCenters[i] = vec2(prngUniform01(), prngUniform01());
    }

    // reset the prngSeed
    prngSeed = globalPrngSeed;

    // Generate colors and find nearest cell in a single pass
    for(int i = 0; i < nVoronoiCells; i++) {
        vec3 currentColor = randomColor();  // Generate color for current cell
        float dist = length(texCoord - cellCenters[i]);
        if(dist < minDist) {
            minDist = dist;
            cellColor = currentColor;  // Use color if this is the nearest cell
        }
    }

    return cellColor;
#endif
    return vec3(0.0, 0.0, 1.0);  // Default return value if SOLUTION_TEXTURING is
not defined
}

// Experiment 6
// Color interpolation with texture support
vec3 getInterpVertexColor(Vertex interpVertex, int textureType)
{
    #ifdef SOLUTION_TEXTURING
    // Choose appropriate texture based on type
    if (textureType == TEXTURE_CHECKERBOARD) {
        return textureCheckerboard(interpVertex.texCoord);
    } else if (textureType == TEXTURE_POLKADOT) {
        return texturePolkadot(interpVertex.texCoord);
    } else if (textureType == TEXTURE_VORONOI) {
        return textureVoronoi(interpVertex.texCoord);
    }
    return interpVertex.color;
    #else
    return interpVertex.color;
    #endif
    return vec3(1.0, 0.0, 1.0);
}

// Experiment 4
// Draws a polygon by projecting, clipping, ratserizing and interpolating it
void drawPolygon(
  vec2 point,
  Polygon clipWindow,
  Polygon oldPolygon,
  inout vec3 color,
  inout float depth)
{
    Polygon projectedPolygon;
    projectPolygon(projectedPolygon, oldPolygon);
```

```
    Polygon clippedPolygon;
    sutherlandHodgmanClip(projectedPolygon, clipWindow, clippedPolygon);

    if (isPointInPolygon(point, clippedPolygon)) {

        Vertex interpolatedVertex = interpolateVertex(point, projectedPolygon);
#ifdef SOLUTION_ZBUFFERING
        // TODO: Implement depth testing
        // Perform depth test
        float newDepth = interpolatedVertex.position.z;
        if (newDepth < depth) {
            // Update color and depth if the new point is closer
            color = getInterpVertexColor(interpolatedVertex,
oldPolygon.textureType);
            depth = newDepth;
        }
#else
      color = getInterpVertexColor(interpolatedVertex, oldPolygon.textureType);
      depth = interpolatedVertex.position.z;
#endif
   }

   if (isPointOnPolygonVertex(point, clippedPolygon)) {
        color = vec3(1);
   }
}
// Main function calls

void drawScene(vec2 pixelCoord, inout vec3 color) {
    color = vec3(0.3, 0.3, 0.3);

    // Convert from GL pixel coordinates 0..N-1 to our screen coordinates -1..1
    vec2 point = 2.0 * pixelCoord / vec2(viewport) - vec2(1.0);

    Polygon clipWindow;
    clipWindow.vertices[0].position = vec4(-0.65,  0.95, 1.0, 1.0);
    clipWindow.vertices[1].position = vec4( 0.65,  0.75, 1.0, 1.0);
    clipWindow.vertices[2].position = vec4( 0.75, -0.65, 1.0, 1.0);
    clipWindow.vertices[3].position = vec4(-0.75, -0.85, 1.0, 1.0);
    clipWindow.vertexCount = 4;

    clipWindow.textureType = TEXTURE_NONE;

    // Draw the area outside the clip region to be dark
    color = isPointInPolygon(point, clipWindow) ? vec3(0.5) : color;

    const int triangleCount = 3;
    Polygon triangles[triangleCount];

    triangles[0].vertexCount = 3;
    triangles[0].vertices[0].position = vec4(-3, -2, 0.0, 1.0);
    triangles[0].vertices[1].position = vec4(4, 0, 3.0, 1.0);
    triangles[0].vertices[2].position = vec4(-1, 2, 0.0, 1.0);
    triangles[0].vertices[0].color = vec3(1.0, 1.0, 0.2);
    triangles[0].vertices[1].color = vec3(0.8, 0.8, 0.8);
```

```
    triangles[0].vertices[2].color = vec3(0.5, 0.2, 0.5);
    triangles[0].vertices[0].texCoord = vec2(0.0, 0.0);
    triangles[0].vertices[1].texCoord = vec2(0.0, 1.0);
    triangles[0].vertices[2].texCoord = vec2(1.0, 0.0);
    triangles[0].textureType = TEXTURE_CHECKERBOARD;

    triangles[1].vertexCount = 3;
    triangles[1].vertices[0].position = vec4(3.0, 2.0, -2.0, 1.0);
    triangles[1].vertices[2].position = vec4(0.0, -2.0, 3.0, 1.0);
    triangles[1].vertices[1].position = vec4(-1.0, 2.0, 4.0, 1.0);
    triangles[1].vertices[1].color = vec3(0.2, 1.0, 0.1);
    triangles[1].vertices[2].color = vec3(1.0, 1.0, 1.0);
    triangles[1].vertices[0].color = vec3(0.1, 0.2, 1.0);
    triangles[1].vertices[0].texCoord = vec2(0.0, 0.0);
    triangles[1].vertices[1].texCoord = vec2(0.0, 1.0);
    triangles[1].vertices[2].texCoord = vec2(1.0, 0.0);
    triangles[1].textureType = TEXTURE_POLKADOT;

    triangles[2].vertexCount = 3;
    triangles[2].vertices[0].position = vec4(-1.0, -2.0, 0.0, 1.0);
    triangles[2].vertices[1].position = vec4(-4.0, 2.0, 0.0, 1.0);
    triangles[2].vertices[2].position = vec4(-4.0, -2.0, 0.0, 1.0);
    triangles[2].vertices[1].color = vec3(0.2, 1.0, 0.1);
    triangles[2].vertices[2].color = vec3(1.0, 1.0, 1.0);
    triangles[2].vertices[0].color = vec3(0.1, 0.2, 1.0);
    triangles[2].vertices[0].texCoord = vec2(0.0, 0.0);
    triangles[2].vertices[1].texCoord = vec2(0.0, 1.0);
    triangles[2].vertices[2].texCoord = vec2(1.0, 0.0);
    triangles[2].textureType = TEXTURE_VORONOI;

    float depth = 10000.0;
    // Project and draw all the triangles
    for (int i = 0; i < triangleCount; i++) {
        drawPolygon(point, clipWindow, triangles[i], color, depth);
    }
}

// Experiment 5
void main() {
    vec3 color = vec3(0);

#ifdef SOLUTION_AALIAS
    // Define constants for quality control
    #define FAST_MODE  // Comment this line to use high quality mode

    #ifdef FAST_MODE
        // Fast mode: 2x MSAA implementation
        const int numSamples = 2;
        vec2 sampleOffsets[numSamples];
        // Standard 2x MSAA sample pattern
        // Rotated grid pattern for better coverage
        sampleOffsets[0] = vec2(-0.25, 0.25);  // Upper left
        sampleOffsets[1] = vec2(0.25, -0.25);  // Lower right
    #else

        // High quality mode: 8x MSAA implementation
```

```
        const int numSamples = 8;
        vec2 sampleOffsets[numSamples];
        // Standard 8x MSAA sample pattern
        // Using rotated grid pattern (similar to DX11 standard pattern)
        sampleOffsets[0]  = vec2( 0.0625, -0.1875);  // Sample 1
        sampleOffsets[1]  = vec2(-0.0625,  0.1875);  // Sample 2
        sampleOffsets[2]  = vec2( 0.3125,  0.0625);  // Sample 3
        sampleOffsets[3]  = vec2(-0.1875, -0.3125);  // Sample 4
        sampleOffsets[4]  = vec2(-0.3125,  0.3125);  // Sample 5
        sampleOffsets[5]  = vec2( 0.1875,  0.3125);  // Sample 6
        sampleOffsets[6]  = vec2( 0.3125, -0.1875);  // Sample 7
        sampleOffsets[7]  = vec2(-0.3125, -0.0625);  // Sample 8
    #endif

    // Accumulate color from samples
    vec3 accumColor = vec3(0.0);

    // Sample multiple points around the current pixel
    for (int i = 0; i < numSamples; i++) {
        vec2 samplePos = gl_FragCoord.xy + sampleOffsets[i];

        // Get color for this sample
        vec3 sampleColor;
        drawScene(samplePos, sampleColor);
        accumColor += sampleColor;
    }

    // Average the accumulated samples
    color = accumColor / float(numSamples);

#else
    drawScene(gl_FragCoord.xy, color);
#endif

    gl_FragColor.rgb = color;
    gl_FragColor.a = 1.0;
}
```