计算机图形学作业1 - 光栅化器

在本次作业中，你将实现一个简单的光栅化器，包括绘制三角形、超采样、层次变换和抗锯齿纹理映射等功能。完成后，你将拥有一个实用的矢量图形渲染器，相当于互联网上广泛使用的 SVG（可缩放矢量图形）文件的简化版本。

请大家自行完成编程，作业中遇到问题，可与其他同学进行技术交流，但请勿直接抄袭代码。

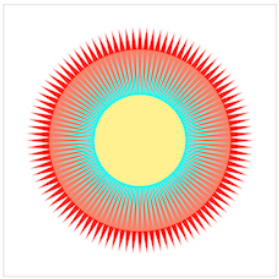
作业结构

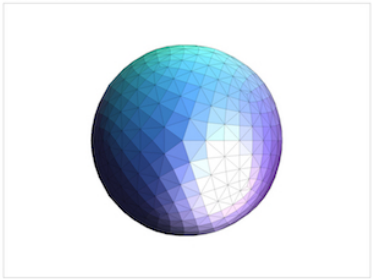
本次作业包括6个部分，覆盖了第 2-6 讲的内容。

入门指南

- 从作业文件夹中找到源代码文件夹 `/code` 。
- 源代码的开发和编译方法见附件 `计算机图形学作业编译说明.pdf` 。
- 在实现代码的同时，请同时完成作业报告。我们提供了写作模板，见附件 `作业报告.docx` ，请按照附加提示完成报告。关于截图和图像格式：GUI 保存的截图默认是 PNG 格式。请不要将 PNG 转换为 JPG 或其他格式，否则会产生压缩伪影。PNG 是无损压缩，能保证图像的质量。

- 代码中使用CGL库进行向量和矩阵运算，使用方法见 `CGL Vectors Library.pdf` 。





如何运行可执行文件

本作业代码编译后生成可执行文件 `draw` ，你可以运行如下命令启动程序：

```
./draw [path to svg file/folder to render]
```

运行时需要指定载入的svg文件路径。我们提供了若干个svg文件，例如你可运行:

```
./draw ../svg/basic/test1.svg
```

启动后你会看到一个由蓝点组成的花朵，这是起始代码提供的点和线的光栅化结果。大部分 SVG 文件需要你完成作业后才能正确渲染。

常用快捷键如下：

键	功能
space	返回初始视角
-	降低采样率
=	提高采样率
Z	切换像素检查器
P	切换纹理像素采样方法
L	切换mipmap层级采样方法
S	当前绘制结果保存为*PNG*图像
1 - 9	切换加载目录中的不同SVG文件

程序 `draw` 的输入参数可以是单个文件或者是一个文件夹（包含了多个*svg*文件）。

```
./draw ../svg/basic/
```

如果不超过9个文件，那可以使用数字键1-9来切换不同的svg文件。

开始熟悉代码

你的主要修改工作会集中在 `rasterizer.cpp` , `transforms.cpp` and `texture.cpp` 这几个文件中。

除此之外，在完成作业的过程中，你还需要理解其他一些源文件和头文件。例如，本次作业（以及后续作业）都使用了 CGL 库。在本次作业中，你可能需要熟悉以下文件中定义的类：`vector2D.h`，`matrix3x3.h` and `color.h`。

下面是运行`draw`程序时的简要流程:

1. 类`SVGParser`(位于`svgparser.h/cpp`)读取输入的*svg*文件。
2. 它启动`OpenGL Viewer`，该viewer包含了一个`DrawRend`渲染器(位于`drawrend.h/cpp`)，该渲染器会无限循环，等待来自鼠标和键盘的输入。
3. 在`DrawRend::redraw()`函数中，高层的绘制工作由不同的`SVGElement`子类(位于`svg.h/cpp`)完成，这些子类随后将其底层的点、线和三角形光栅化数据传递给`Rasterizer`类的相应方法。

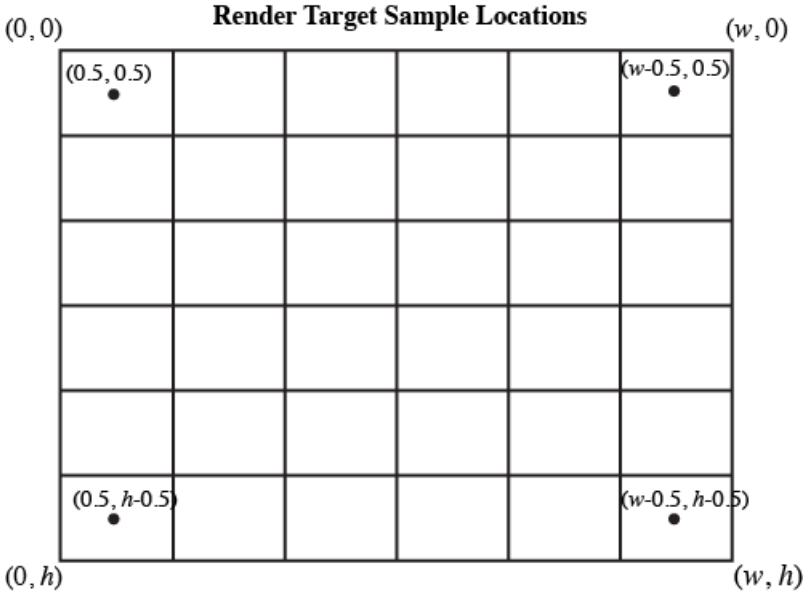
一个简单示例：绘制点

起始代码已经实现了 绘制二维点 的功能。要理解它的工作方式，可以先查看`SVG::draw()`函数（位于`svg.h`文件中）。

1. SVG对象会通过依次调用每个元素的`draw()`函数来绘制 SVG 文件中的所有元素。
2. 每种元素类型都会调用`Rasterizer`对象中相应的绘制函数。
    - 以`Point`元素类型为例，`Point::draw()`最终会调用`RasterizerImp::rasterize_point()`（定义在`rasterizer.cpp`）里实现的具体绘制函数。SVG 文件中元素的位置是用 局部坐标系 定义的，因此`Point::draw()`会先把点的位置转换到屏幕空间 (screen space) 坐标，然后再传递给`RasterizerImp::rasterize_point()`。

函数`RasterizerImp::rasterize_point()`负责真正地把点绘制出来。在本作业中，屏幕空间的定义如下（假设输出图像大小为`(target_w, target_h)`）：

- `(0, 0)`坐标对应输出图像的左上角；
- `(target_w, target_h)`坐标对应输出图像的右下角；
- 请注意屏幕采样点取值为半整数值，如左上角采样点坐标为（0.5，0.5），右下角采样点坐标应为（target_w-0.5, target_h-0.5）。



在进行点的光栅化时，我们采用以下规则： 一个点最多覆盖一个屏幕采样点，也就是距离该点最近的那个屏幕空间采样点。假设点在屏幕空间的位置是 (x, y)，其实现方式如下：

```
int sx = (int) floor(x);
int sy = (int) floor(y);
```

当然，代码在访问渲染目标缓冲区时，不能修改 无效的像素位置。因此需要进行边界检查：

```
if ( sx < 0 || sx >= target_w ) return;
if ( sy < 0 || sy >= target_h ) return;
```

如果点的位置确实落在屏幕范围内，我们就用该点的 RGB 颜色值 填充对应的像素：

```
  rgb_framebuffer_target[3 * (y * width + x)] = (unsigned char)(c.r * 255
  rgb_framebuffer_target[3 * (y * width + x) + 1] = (unsigned char)(c.g *
  rgb_framebuffer_target[3 * (y * width + x) + 2] = (unsigned char)(c.b *
```

注意：在本次作业中，我们 不支持半透明或 alpha 混合，尽管这些是 SVG 文件格式本身支持的特性。

作业任务

任务 1：绘制单色三角形 (20 分)

相关课程讲义：第 2 讲

在本任务中，你需要实现 `rasterize_triangle` 函数，其定义在 `rasterizer.cpp` 文件中。你的代码实现应该满足以下要求：

- 使用课堂上讲解的采样方法来对三角形进行光栅化。
- 每个像素只采样一次，采样点应位于像素的中心，而不是角落。也就是说，采样点的坐标应为： (整数坐标 + 0.5, 整数坐标 + 0.5)。
- 在 第 2 部分中你会实现子像素级的超采样。但在这里，你只需对每个像素采样一次，并调用 `fill_pixel()` 辅助函数（可参考起始代码中的 `rasterize_point` 示例。
- 你的实现应当默认：位于三角形边界上的采样点应被绘制。鼓励（但不强制）实现 OpenGL 的边界规则，以处理采样点恰好落在边上的情况。确保三角形的边缘不会出现漏绘现象。
- 不要简单地对整个帧缓冲区逐像素检查，如此太低效。你的实现效率至少应与“只在三角形的包围盒 (bounding box) 内采样”一样高。
- 你的代码应当能正确绘制任意顶点顺序的三角形（顺时针或逆时针）。参见 `svg/basic/test6.svg` 。

完成后，你应当能够正确渲染包含 单色多边形 的测试用 SVG 文件。这些多边形在传入你的函数前会被拆分为三角形。以下文件应能正确渲染： `basic/test3.svg` , `basic/test4.svg` , `basic/test5.svg` , and `basic/test6.svg` 。

需要修改的函数包括：

1. `RasterizerImp::rasterize_triangle()` ，位于 `rasterizer.cpp` 。

额外加分点：如果你能让三角形光栅化器运行得更快（例如：把冗余算术运算移出循环、减少内存访问、避免对包围盒中的每个采样点逐一检查），可以获得加分。请在报告中写明你所使用的优化方法。可以使

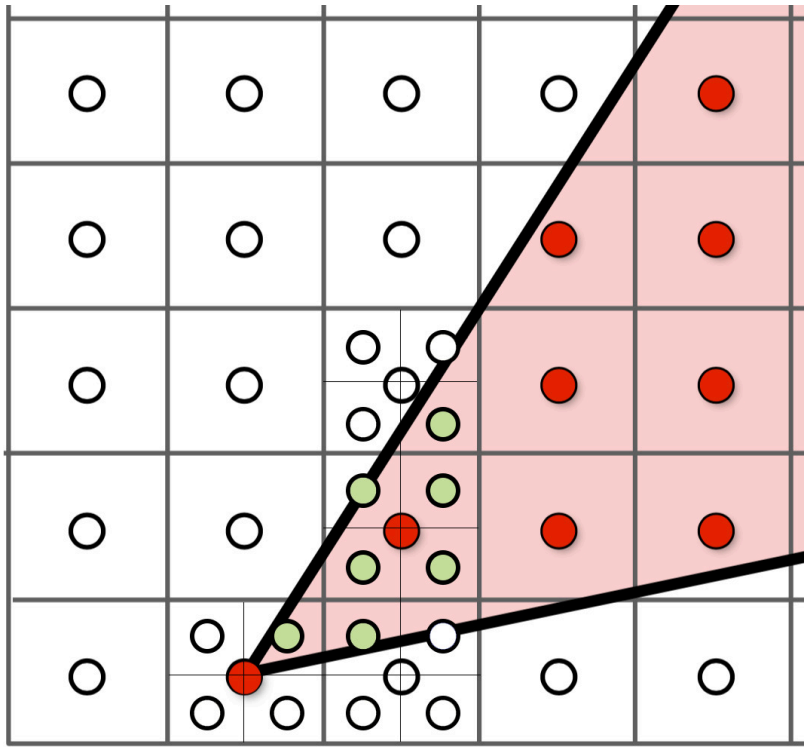
用 `clock()` 或 `std::chrono::high_resolution_clock` 来对比基础实现和优化实现的运行时间。

任务 2：通过超采样实现反走样 (20 分)

相关课程讲义：第 3 讲

在本任务中，你需要使用 超采样 (supersampling) 来对三角形进行抗锯齿处理。`DrawRend` 类中的 `sample_rate` 参数决定了每个像素要采样多少次。（可通过键盘 `-` 和 `=` 调整）

下图展示了：当每个像素采样 4 次时，结果比只采样一次要平滑得多。位于三角形内部的子采样点的比例越高，三角形边缘就会越平滑。



超采样的实现，请在像素区域内，按照 `sqrt(sample_rate) * sqrt(sample_rate)` 的网格分布进行采样。（sample_rate 是 RasterizerImp 类的一个成员变量。）

一种直观的理解方式是：先渲染一个 更高分辨率的图像，然后再把它下采样到帧缓冲区的目标分辨率。

在 任务 1 中的 `fill_pixel` 函数是直接在帧缓冲区上绘制的。但对于超采样，你应当先绘制到 sample_buffer，即将所有子采样的结果填充到与输出像素对应的子采样单元里。

光栅化器的整体流程回顾:

1. SVGParser 解析 SVG 文件并生成 SVG 类的表示。
2. 当光栅化开始时，渲染器 (`DrawRend::redraw`) 调用 SVG::draw。
3. `SVG::draw` 调用具体的点/线/三角形光栅化函数，逐个图元生成图像。
4. `DrawRend::redraw` 调用线段光栅化函数绘制边框。
5. `DrawRend::redraw` 调用 `RasterizerImp::resolve_to_framebuffer()` ，将光栅化器的内部缓冲区内容转换到屏幕缓冲区，以便显示并写入文件。

实现提示：

- 你需要分配合适的内存来存储超采样数据。推荐使用`RasterizerImp::sample_buffer`向量（见`rasterizer.h`）。采样缓冲区的大小取决于 帧缓冲区的尺寸（窗口大小变化时会改变）和 超采样率（按键调整）。你需要动态更新其大小。
- 在每一帧绘制开始时，应当清空采样缓冲区和/或帧缓冲区，以便擦除上一次的绘制结果。
- 更新`rasterize_triangle`，使其将子采样结果填入 sample_buffer。你可以自行设计如何组织 supersample buffer 中的数据。把 supersample buffer 理解为一个高分辨率帧缓冲区：
  - 例如，若采用 4x4 超采样，且目标帧缓冲区大小为 1000x1000，那么实际上就是在 4000x4000 的高分辨率图像上渲染场景。完成渲染后，再把每个 4x4 的子采样结果平均，得到目标像素的最终颜色。这样需要更多内存来存储高分辨率的结果。思考一下：能否在不额外占用内存的情况下得到相同结果？这样会带来哪些工程上的权衡?
- 在场景中所有元素光栅化完成后，你需要将 supersample buffer 的内容写入帧缓冲区。 这一步通常称为 resolve supersamples。**`RasterizerImp::resolve_to_framebuffer`**就是完成这一工作的函数，它在**in `drawrend.cpp`** drawrend.cpp 的最后一步被调用。
- 注意颜色数据类型的转换：`RasterizerImp::rgb_framebuffer_target`指向最终显示的帧缓冲区像素数据。`rgb_framebuffer_target`其存储格式是 8 位整数数组（每个像素的 R/G/B）。这是大多数实际图形系统要求的格式。相比之下，`RasterizerImp::sample_buffer`（建议用于 supersampling 内存）是`Color`对象数组，其中 R/G/B 值用浮点数存储。你需要在两种数据类型之间做转换，注意避免浮点转整数时的 舍入或溢出错误。
- 在图形学中，颜色可以看作与向量数据相同的东西：它是 3 维向量（如果有 alpha 通道就是 4 维）。这意味着，一个三维向量 (x, y, z) 不仅可以表示点或方向，也可以表示颜色。课程后续有时间的话会介绍颜色空间与色彩理论。
- 在实现 supersampling 后，你可能会发现 点和线 无法正确绘制。点和线本身不是 supersampled，但它们仍需要写入 supersample buffer。如果需要，请修改`RasterizerImp::fill_pixel`来恢复其功能。一种思路是：把某个点或线对应的所有子采样位置都填充为同一个颜色，这样下采样后仍然显示为一个像素。注意：你不需要对点和线做抗锯齿处理。

当实现完成后，如果你使用大于 1 的采样率，三角形的边缘应当显著变得更加平滑！你可以使用 像素检查器 来仔细观察采样率差异（快捷键见前文）。注意：切换到较高采样率时，可能需要等待数秒。

再次提示下，需要修改的函数包括：

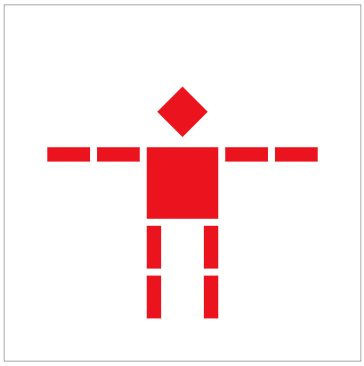
1. 管理 supersample buffer 内存：`RasterizerImp::sample_buffer`，`RasterizerImp::set_sample_rate()`，`RasterizerImp::set_framebuffer_target()`，`RasterizerImp::clear_buffers()`，在文件`rasterizer.h/cpp`。
2. 实现三角形 supersampling：`RasterizerImp::rasterize_triangle()`，`RasterizerImp::fill_pixel()`，在文件`rasterizer.cpp`。
3. 解析 supersamples 到帧缓冲区：`RasterizerImp::resolve_to_framebuffer()`。

任务 3：几何变换 (10 分)

相关课程讲义：第 4 讲

在本任务中，你需要根据SVG规范，在*transforms.cpp*文件中实现三种几何变换。这些矩阵是 3×3 的，因为它们在 齐次坐标 (homogeneous coordinates) 中运算。你可以查看同一文件中 `*` 运算符的重载方式，来理解这些变换矩阵是如何作用在 `Vector2D` 实例上的。

完成这三种变换后，文件*svg/transforms/robot.svg*应能正确渲染，如下图所示：



需要修改的函数在*transforms.cpp*文件中：

1. `translate`
2. `scale`
3. `rotate`

额外加分点：为 GUI 添加一个额外功能。例如，你可以绑定两个未使用的按键来旋转视口 (viewport)。请保存一张示例图片来展示你实现的功能，并在报告中说明：你是如何修改 SVG → NDC（归一化设备坐标）和 NDC → 屏幕空间 的矩阵堆栈，来完成这个功能的。

任务 4：质心坐标 (10 分)

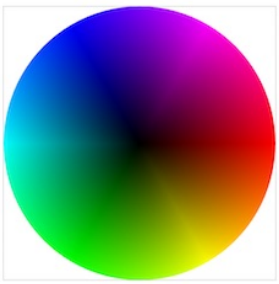
相关课程讲义：第 5 讲

在本任务中，你需要实现 `RasterizerImp::rasterize_interpolated_color_triangle(...)`，用于绘制一个三角形：三角形的三个顶点分别定义了颜色；在三角形内部区域使用质心插值 (barycentric interpolation) 来对颜色进行插值。

完成本部分后，你应当能够在*svg/basic/test7.svg*中看到一个色轮（如下图右侧所示）。

需要修改的函数：

1. `RasterizerImp::rasterize_interpolated_color_triangle(...)`



任务 5：基于像素采样的纹理映射 (15 分)

相关课程讲义：第 6 讲

在本任务中，你需要实现`RasterizerImp::rasterize_textured_triangle(...)`，绘制一个带有纹理映射的三角形：三个顶点分别带有二维纹理坐标 (UV coordinates)；使用给定的纹理图像，在三角形内部通过纹理采样来确定颜色。在本任务中，你需要在完整分辨率的纹理图像上实现两种采样方法（如课程讲义所述）：最近邻采样 (nearest neighbor sampling)和双线性插值 (bilinear interpolation)。

在 GUI 中，可以通过按下`'P'`键切换`RasterizerImp`的`PixelSampleMethod`变量`psm`。当`psm == P_NEAREST`时，使用 最近邻采样；当`psm == P_LINEAR`时，使用双线性插值采样。你需要在以下函数中实现逻辑，`Texture::sample_nearest`和`Texture::sample_bilinear`，并在`RasterizerImp::rasterize_textured_triangle(...)`中调用它们。这样进行抽象，你的实现也能在任务 6（三线性 mipmap 过滤） 中复用。

完成本任务后，你应当能够正确光栅化*svg/texmap/*文件夹下依赖纹理映射的*svg*文件了。

注意事项：

- `Texture`结构体（定义在`texture.h`）存储了一个 mipmap 层级结构（如讲义中所述），保存在`mipmap`变量中。每个纹理图像存储为`MipLevel`类型的对象。
- `MipLevel::texels`以常见的 RGB 格式 存储纹理图像像素（与帧缓冲区像素存储方式相同）。
- `MipLevel::get_texel(...)`函数可能对你有帮助。
- 在本作业的这一部分，你还 不需要实现层级采样 (mipmapping)，因此程序应默认使用 第 0 层 (full resolution) 的纹理图像。

需要修改的函数：

1. `RasterizerImp::rasterize_textured_triangle`
2. `Texture::sample_nearest`
3. `Texture::sample_bilinear`

任务 6：基于 mipmap 的纹理映射 (25 分)

相关课程讲义：第 6 讲

在这一部分，你需要更新`RasterizerImp::rasterize_textured_triangle(...)`，使其支持从不同的 mipmap 层 (`MipLevel`) 中进行采样。在 GUI 中，可以通过按下`L`键来切换`RasterizerImp`的`LevelSampleMethod`变量`lsm`。你需要在`Texture::sample`函数中实现以下采样方式：

- 当`lsm == L_ZERO`，从 第 0 层`MipLevel`采样（和任务 5 相同）。
- 当`lsm == L_NEAREST`，计算最接近的 mipmap 层级，将该层作为参数传递给 nearest 或 bilinear 采样函数。
- 当`lsm == L_LINEAR`，计算一个连续的 mipmap 层级（小数值），对相邻的两个 mipmap 层分别采样，并做加权平均（即讲义中介绍的线性插值方法）。

你将实现`Texture::get_level`函数，来获取mipmap层级。这里面你将会使用到$(\frac{du}{dx}, \frac{dv}{dx})$和$(\frac{du}{dy}, \frac{dv}{dy})$来计算正确的mipmap层级。为此，你必须在对三角形内的某个顶点(x, y)进行如下计算：

1. 计算该点及其相邻点，(x, y), $(x+1, y)$, and $(x, y+1)$的*uv*质心坐标，`sp.p_uv`，`sp.p_dx_uv`, and `sp.p_dy_uv`，将这些值保存到`SampleParams`结构体`sp`中，并传给`Texture::get_level`。

2. 在 `Texture::get_level` 内，计算差分向量 `sp.p_dx_uv - sp.p_uv` 和 `sp.p_dy_uv - sp.p_uv` 。
3. 按照全分辨率纹理的宽度和高度 对差分向量进行缩放。

有了这些步骤，你就可以按照讲义中的公式完成 mipmap 层级的计算。

注意事项：

- 变量 `lsm` 和 `psm` 是独立的，可以自由组合。例如： `psm==[P_NEAREST, P_LINEAR] x lsm==[L_ZERO, L_NEAREST, L_LINEAR]` 都是可行的。
- 当 `lsm == L_LINEAR` 且 `psm == P_LINEAR` 时，这种方法称为三线性采样（既三线性纹理滤波）。
- 你可以将 `Texture::get_level` 返回的值归一化（除以最大mipmap层数），再把它作为颜色从 `Texture::sample` 返回。这样缩放图像时，你能直观看到不同区域使用了哪一层 mipmap，非常有助于调试和理解采样原理。
- 注意：不要复制整个 MipLevel！一定要通过指针或引用访问。复制整个 mipmap 层作为参数会非常慢。

需要修改的函数：

1. `RasterizerImp::rasterize_textured_triangle`
2. `Texture::sample`
3. `Texture::get_level`