Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI-4321 Computación Gráfica I
Prof. Alexandra La Cruz

# A Simple SVG Rasterizer

**Autores:**
Carlos Rivero 13-11216
Carlos Sivira 15-11377
José Barrera 15-10123

# TASK 1
## Drawing Triangles

- Dibujado del triángulo en función del cuadrado formado por **maxX**, **minX**, **maxY** y **minY** (que representa el rectángulo de menor tamaño que encierra el triángulo).
- Un pixel o un sample es dibujado si **insideTriangle** es verdadero, es decir, si se encuentra dentro del triángulo. La verificación consiste en comparar con los 3 lados del triángulo, de acuerdo a lo visto en la clase [#2 CS248].

```
void SoftwareRendererImp::rasterize_triangle( float x0, float y0,
                                              float x1, float y1,
                                              float x2, float y2,
                                              Color color ) {
  float maxX = fmax(x0, fmax(x1, x2));
  float minX = fmin(x0, fmin(x1, x2));

  float maxY = fmax(y0, fmax(y1, y2));
  float minY = fmin(y0, fmin(y1, y2));

  double invSampleRate = 1.0 / sample_rate;
  double halveInvSampleRate = invSampleRate / 2;

  float stepX, stepY;

  for(int x = minX; x < maxX; x++) {
    for (int y = minY; y < maxY; y++) {

      for (int sx = 0; sx < sample_rate; sx++) {
        stepX = halveInvSampleRate + invSampleRate*sx;

        for (int sy = 0; sy < sample_rate; sy++) {
          stepY = halveInvSampleRate + invSampleRate*sy;
          if (insideTriangle(x + stepX, y + stepY, x0, y0, x1, y1, x2, y2)){
            int sampleX = sx + sy * sample_rate;  // sample number
            int sampleY = x + y * target_w;       // pixel number

            fill_sample(sampleX, sampleY, color);

          }
        }
      }
    }
  }
}
```

# TASK 1
## Drawing Triangles

- La verificación consiste en comparar con los 3 lados del triángulo y ver si está del lado correcto, de acuerdo a lo visto en la clase #2 CS248.
- Como los puntos del triángulo no siempre están ordenados de la misma manera, la función retorna si el punto está completamente contenido indiferentemente del orden de los vértices del triángulo.

```
bool insideTriangle(float x, float y,
                    float x0, float y0,
                    float x1, float y1,
                    float x2, float y2) {

  float dy = y1 - y0;
  float dx = x1 - x0;
  float c = y0 * (x1 - x0) - x0 * (y1 - y0);
  float l1 = dy*x - dx*y + c;

  dy = y2 - y1;
  dx = x2 - x1;
  c = y1 * (x2 - x1) - x1 * (y2 - y1);
  float l2 = dy*x - dx*y + c;

  dy = y0 - y2;
  dx = x0 - x2;
  c = y2 * (x0 - x2) - x2 * (y0 - y2);
  float l3 = dy*x - dx*y + c;

  return (l1 <= 0 && l2 <= 0 && l3 <= 0) || (l1 >= 0 && l2 >= 0 && l3 >= 0);
}
```

# TASK 2
## Anti-Aliasing using Supersampling

- Inicialmente se asigna el espacio de memoria para el buffer de samples. Se inicializa cada sample en blanco para evitar problemas de refrescado entre SVGs.
- **Resolve** genera un color que resulta de calcular el promedio de los samples asociados a cada pixel y colorea el pixel de ese color.
- Finalmente, se libera la memoria del buffer de samples.

```
94 void SoftwareRendererImp::draw_svg( SVG& svg ) {
95
96   // set top level transformation
97   transformation = canvas_to_screen;
98
99   supersample_target = (unsigned char*) malloc(4 * target_h * target_w * sample_rate * samp
   le_rate * sizeof(uint8_t));
>>   memset(supersample_target, 255, 4 * target_w * target_h * sample_rate * sample_rate);
101
102 +-- 17 lines: draw all elements----------------------------------------------------------
119   resolve();
120
121 }
```

```
// resolve samples to render target
void SoftwareRendererImp::resolve( void ) {

  int sampleRateSquared = sample_rate * sample_rate;

  for(int x = 0; x < target_w; x++) {
    for (int y = 0; y < target_h; y++) {
      int r = 0;
      int g = 0;
      int b = 0;
      int a = 0;

      for (int sx = 0; sx < sampleRateSquared; sx++) {
        r += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + sx * 4];
        g += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 1];
        b += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 2];
        a += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 3];
      }

      render_target[4 * (x + y * target_w)] = r/sampleRateSquared;
      render_target[4 * (x + y * target_w) + 1] = g/sampleRateSquared;
      render_target[4 * (x + y * target_w) + 2] = b/sampleRateSquared;
      render_target[4 * (x + y * target_w) + 3] = a/sampleRateSquared;

    }
  }

  free(supersample_target);
}
```

# TASK 2
## Anti-Aliasing using Supersampling

- Para soportar supersampling, se modifica **fill_pixel** para que tome en cuenta cada sample asociado a un pixel.
- En **fill_sample, sx** representa el sample actual a colorear y **sy** representa el píxel al cual pertenece dicho sample. Se hace de esta manera puesto que el buffer se samples guarda de forma consecutiva todos los samples asociados a un pixel. Para n pixeles (P) y k samples (S) se tiene:

Buffer de pixels [P0, P1, …, Pn]
Buffer de samples [S00,...S0k, S10,... ,Snk]

```cpp
// fill a sample location with color
void SoftwareRendererImp::fill_sample(int sx, int sy, const Color &color) {
  // sx = sample number
  // sy = pixel number

  float inv255 = 1.0 / 255.0;
  int sampleRateSquared = sample_rate * sample_rate;
  if (sx < 0 || sx >= sampleRateSquared) return;
  if (sy < 0 || sy >= target_w*target_h) return;

  Color sample_color;
  sample_color.r = supersample_target[4 * sampleRateSquared * sy + sx*4] * inv255;
  sample_color.g = supersample_target[4 * sampleRateSquared * sy + sx*4 + 1] * inv255;
  sample_color.b = supersample_target[4 * sampleRateSquared * sy + sx*4 + 2] * inv255;
  sample_color.a = supersample_target[4 * sampleRateSquared * sy + sx*4 + 3] * inv255;

  sample_color = alpha_blending(sample_color, color);

  supersample_target[4 * sampleRateSquared * sy + sx*4] = (uint8_t)(sample_color.r * 255);
  supersample_target[4 * sampleRateSquared * sy + sx*4 + 1] = (uint8_t)(sample_color.g * 255);
  supersample_target[4 * sampleRateSquared * sy + sx*4 + 2] = (uint8_t)(sample_color.b * 255);
  supersample_target[4 * sampleRateSquared * sy + sx*4 + 3] = (uint8_t)(sample_color.a * 255);
}
```

```cpp
// fill samples in the entire pixel specified by pixel coordinates
void SoftwareRendererImp::fill_pixel(int x, int y, const Color &color) {

  int sampleRateSquared = sample_rate * sample_rate;

  if (x < 0 || x >= target_w) return;
  if (y < 0 || y >= target_h) return;

  Color pixel_color;
  float inv255 = 1.0 / 255.0;
  int r = 0;
  int g = 0;
  int b = 0;
  int a = 0;
  for (int sx = 0; sx < sampleRateSquared; sx++) {
    r += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + sx * 4];
    g += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 1];
    b += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 2];
    a += supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 3];
  }
  pixel_color.r = r/sampleRateSquared * inv255;
  pixel_color.g = g/sampleRateSquared * inv255;
  pixel_color.b = b/sampleRateSquared * inv255;
  pixel_color.a = a/sampleRateSquared * inv255;

  pixel_color = alpha_blending(pixel_color, color);

  for (int sx = 0; sx < sampleRateSquared; sx++) {
    supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + sx * 4] = (uint8_t)(pixel_color.r * 255);
    supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 1] = (uint8_t)(pixel_color.g * 255);
    supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 2] = (uint8_t)(pixel_color.b * 255);
    supersample_target[ 4 * sampleRateSquared * (x + y * target_w) + (sx * 4) + 3] = (uint8_t)(pixel_color.a * 255);
  }
}
```

# TASK 3

## Modeling and viewing transforms

- 3.1
  - Para respetar el marco de referencia para cada transformación, se salva la transformación de mayor jerarquía y se procede a dibujar cada elemento operando su transformación interna con la de su marco de referencia.
  - Al terminar el dibujado del elemento, se restaura la transformación asociada al marco de referencia de mayor jerarquía que aún tiene elementos sin dibujar.
- 3.2
  - Se aplican las transformaciones de traslación y escalado del viewbox creando su respectiva matriz de transformación, estas se multiplican en el orden deseado.

Fuente

```
125  void SoftwareRendererImp::draw_element( SVGElement* element ) {
126
127    Matrix3x3 transform_save = transformation;
128
129    transformation = transformation * element->transform;
130
131  >  switch (element->type) {…
159
160    transformation = transform_save;
161
162  }
```

```
void ViewportImp::set_viewbox( float x, float y, float span ) {

  Matrix3x3 translation = Matrix3x3::identity();
  Matrix3x3 scale = Matrix3x3::identity();

  translation(0, 2) = span - x;
  translation(1, 2) = span - y;

  scale(0, 0) = 1.0 / (2 * span);
  scale(1, 1) = 1.0 / (2 * span);

  this->x = x;
  this->y = y;
  this->span = span;

  set_canvas_to_norm(scale * translation);

  this->x = x;
  this->y = y;
  this->span = span;
}
```

# TASK 4
## Drawing Scaled Images

- F
- Para **sample_nearest,** primero se revisa que el nivel sea válido, luego se ubica el punto dentro del mipmap y se devuelve el color del texel más cercano.
- **Sample_bilinear** no fue implementado (aunque se intentó). Para su desarrollo se requiere tomar 4 samples más cercanos a un punto dentro del mipmap. Se devuelve la interpolación lineal de estos 4 samples.
- **Rasterize_image** no fue implementado (aunque se pensó). Funcionaría parecido a **rasterize_triangle**, salvo que antes de dibujar el pixel, se debe calcular el color correspondiente haciendo **sample_nearest** o **sample_bilinear** pasando como argumento el punto en la textura asociado al sample.

```
Color Sampler2DImp::sample_nearest(Texture& tex,
                                   float u, float v,
                                   int level) {

  // return magenta for invalid level
  if (level >= tex.mipmap.size())
    return Color(1,0,1,1);

  // Task 4: Implement nearest neighbour interpolation
  MipLevel mLevel = tex.mipmap[level];

  int width = mLevel.width;
  int height = mLevel.height;
  float x = floor(u * width);
  float y = floor(v * height);

  Color color;
  color.r = mLevel.texels[4 * (x + y * width)] / 255.f;
  color.g = mLevel.texels[4 * (x + y * width) + 1] / 255.f;
  color.b = mLevel.texels[4 * (x + y * width) + 2] / 255.f;
  color.a = mLevel.texels[4 * (x + y * width) + 3] / 255.f;

  return color;
}
```

# TASK 5
## Alpha Compositing

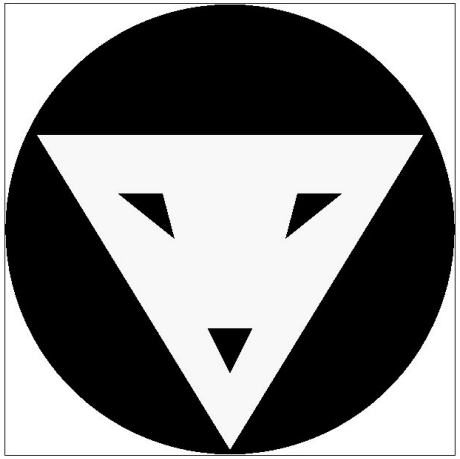
- La implementación está basada en la sugerencia indicada en el proyecto.
- Dado dos colores (el del canvas y el del elemento), el algoritmo calcula el inverso del alpha del color del elemento.
- Retorna un nuevo color que se compone del color del elemento más el color del canvas multiplicado con el alpha calculado.

```
Color SoftwareRendererImp::alpha_blending(Color pixel_color, Color color)
{
  Color return_color = color;
  color.r = color.r * color.a;
  color.g = color.g * color.a;
  color.b = color.b * color.a;

  pixel_color.r = pixel_color.r * pixel_color.a;
  pixel_color.g = pixel_color.g * pixel_color.a;
  pixel_color.b = pixel_color.b * pixel_color.a;

  return_color.a = 1 - (1 - color.a) * (1 - pixel_color.a);
  return_color.r = (1 - color.a) * pixel_color.r + color.r;
  return_color.g = (1 - color.a) * pixel_color.g + color.g;
  return_color.b = (1 - color.a) * pixel_color.b + color.b;
  return return_color;
}
```

# TASK 6
## Draw Something!

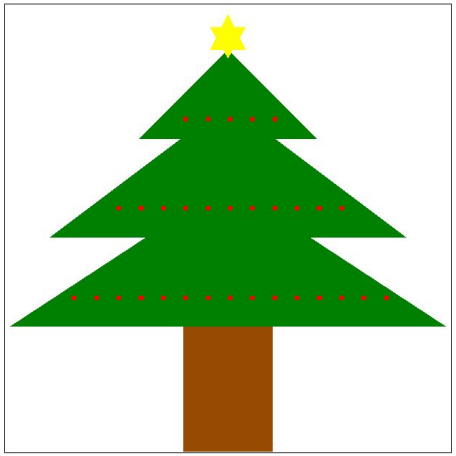
- Se dibujó una cara, usando el elemento polygon, para generar 4 triángulos y 1 círculo, partiendo del ejemplo 03_circle.svg
- Se dibujó un árbol de navidad, usando el elemento polygon y rect, para generar 7 triángulos. y 31 cuadrados.
- Estos se encuentra en la carpeta svg/custom

Software Renderer



Framerate: 60 fps

Software Renderer



Framerate: 59 fps

# Extra
## Drawing Lines

- Se aplicó el algoritmo de Bresenham que permite dibujar una línea sencilla sin suavizado.
- El algoritmo colorea pixel a pixel, desde (x0, y0) hasta (x1,y1), desplazándose a través del eje x y del eje y. Para esto, calcula el error entre la línea real y la línea construida, luego se mueve en el eje x e incrementa el error hasta que este sea mayor que la diferencia de altura entre el punto inicial y el final de la recta, cuando esto ocurre se desplaza sobre el eje y.

```
void SoftwareRendererImp::rasterize_line( float x0, float y0,
                                          float x1, float y1,
                                          Color color) {

  int xx0 = (int)floor(x0);
  int xx1 = (int)floor(x1);
  int yy0 = (int)floor(y0);
  int yy1 = (int)floor(y1);

  int dx = abs(xx1 - xx0);
  int sx = xx0<xx1 ? 1 : -1;
  int dy = -abs(yy1 - yy0);
  int sy = yy0<yy1 ? 1 : -1;
  int err = dx+dy;
  while(true) {
    fill_pixel(xx0, yy0, color);
    if (xx0 == xx1 && yy0 == yy1) break;
    int e2 = 2*err;
    if (e2 >= dy){
      err += dy;
      xx0 += sx;
    }

    if (e2 <= dx) {
      err += dx;
      yy0 += sy;
    }
  }

}
```