Graficacion por Computadora

PRIMER PROYECTO

Descripción General

En este proyecto, se implementará un software de rasterización simple que dibuja puntos, líneas, triángulos e imágenes de mapa de bits. Cuando termines, tendrán un visor que soporta las características básicas del formato Scalable Vector Graphics (SVG).

Primeros pasos

Se distribuirán las tareas con git. Se puede encontrar el repositorio para esta asignación en http://462cmu.github.io/asst1_drawsvg/. Si no está familiarizado con git, aquí es lo que necesita hacer para obtener el código de arranque:

```
$ git clone https://github.com/462cmu/asst1_drawsvg.git
```

Esto creará una carpeta asst1_drawsvg con todos los archivos fuente.

Instrucciones de Compilación

Con el fin de facilitar el proceso de ejecución en distintas plataformas, usaremos CMake para nuestras tareas. Se necesita una instalación de CMake de versión 2.8 o mayor para crear el código para esta tarea. También debería ser relativamente fácil la tarea de construir y trabajar localmente en tu OSX o Linux. Actualmente no se admite la construcción en Windows, pero el proyecto se puede ejecutar por SSH'ing a través de Andrew Linux utilizando MobaXterm.

Si estás trabajando en OS X y no CMake instalado, se recomienda instalar a través de MacPorts :

```
sudo port install cmake
```

O Homebrew:

```
brew install cmake
```

Para compilar el código para esta tarea:

- Crear un directorio para el código:

```
$ cd p1 && mkdir build && cd build
```

- Ejecutar CMake para generar el archivo makefile:

```
$ cmake ../src
```

- Compilar el código:

```
$ make
```

- Instalar el ejecutable en el directorio raíz del proyecto:

```
$ make install
```

Usando la aplicación Mini-SVG Viewer

Cuando se han compilado con éxito el código, se obtendrá un ejecutable llamado **drawsvg**. El ejecutable **drawsvg** toma exactamente un argumento desde la línea de comandos. Puede cargar un solo archivo SVG especificando su ruta. Por ejemplo, para cargar el archivo de ejemplo *svg/basic/test1.svg* :

```
./drawsvg ../svg/basic/test1.svg
```

Al ejecutar la aplicación, se verá un cuadro de una flor hecha de un montón de puntos azules. Este código que dibuja estos puntos es el que se debe modificar.

Mientras se mira a la solución de referencia, mantengan presionado el botón izquierdo del mouse y arrastre el cursor para encuadrar el punto de

vista. También puede utilizar la rueda de desplazamiento para acercar la vista. (Se puede presionar *Espacio* para restaurar la vista a las condiciones por defecto). También puede comparar la salida de su ejecución con la de la implementación de referencia. Para cambiar la vista diff, presionen `D`. También se ha proporcionado un "inspector de pixel" para examinar los detalles a nivel de pixel de la aplicación muestra actual más claramente. El inspector de pixel se enciende con el `Z` clave.

Para mayor comodidad, drawsvg puede también aceptar una ruta a un directorio que contiene varios archivos SVG. Para cargar archivos de svg/basic:

```
./drawsvg ../svg/basic
```

La aplicación cargará hasta nueve archivos de esa ruta y cada archivo se cargará en una pestaña. Se puede cambiar a una pestaña específica con teclas `1` a través de `9`.

Resumen de los controles del visualizador

Una tabla de todos los controles de teclado en la aplicación **dibujar** se proporciona abajo.

Comando	Key
Ir a la pestaña	`1` ~ `9`
Cambiar al renderer de hw	`H`
Cambiar al renderer de sw	`S`
Toggle sw renderer impl (student soln/ref soln)	`R`
Regenerate mipmaps for current tab (student soln)	`;`
Regenerate mipmaps for current tab (ref soln)	`'`
Increase samples per pixel	`=`
Decrease samples per pixel	`-`
Toggle text overlay	`` ` ``

Toggle pixel inspector view	Z
Toggle image diff view	D
Reset el viewport a la posición por default.	SPACE

Lo que necesitan hacer

La asignación se divide en ocho grandes tareas que se describen a continuación en el orden en que se sugiere que se les trate. Por supuesto pueden hacer la asignación en cualquier orden que ustedes elijan. **Se debe tener en cuenta que cumplir los requisitos de las tareas posteriores puede implicar la reestructuración del código que se implementó antes.**

Conociendo el código Inicial

Antes de empezar, aquí hay alguna información básica sobre la estructura del código inicial.

La mayor parte de su trabajo está en implementar SoftwareRendererImp en software_renderer.cpp. El método más importante es draw_svg que acepta un objeto SVG para dibujar. Un archivo SVG define su canvas (que define un espacio de coordenadas 2D) y especifica una lista de elementos de forma (como puntos, líneas, triángulos e imágenes) que deben ser dibujado en ese canvas. Cada elemento de forma tiene un número de parámetros de estilo (por ejemplo, color), así como una transformación de modelado utilizada para determinar la posición del elemento en el canvas. Se puede encontrar la definición de la clase SVG (y todos los SVGElements asociados) en svg.h. Un tipo de SVGElement es un grupo que contiene elementos secundarios. Por lo tanto, usted debe pensar un archivo SVG como definir un árbol de elementos de forma. (Los nodos interiores del árbol son grupos y hojas son formas).

Otro importante método en la clase de SoftwareRendererImp es set_render_target(), el cual proporciona a su código un buffer correspondiente a la imagen de salida (también proporciona anchura y la altura del buffer en píxeles, que se almacenan localmente como target_w y target_h). Este buffer se denomina el "render target" en muchas aplicaciones, ya que es el " target " de los comandos de renderizado. Utilizamos el termino pixel aquí a propósito porque los valores de este buffer son los valores que se mostrarán en pantalla. Valores de los píxeles se almacenan en formato de fila mayor, y cada pixel es un valor RGBA de 8 bits (32 bits en total). En su implementación es necesario que rellene el contenido de este buffer cuando se pide dibujar un archivo SVG.

set_render_target() se llama cuando el usuario cambia el tamaño de la ventana de la aplicación.

Un Ejemplo Simple: Dibujado de puntos

Se le da el código inicial que ya implementa el dibujado de 2D.

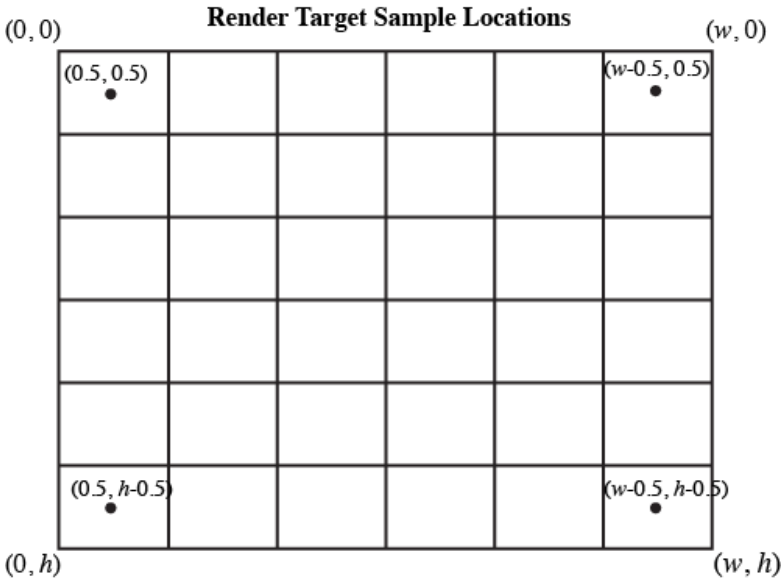
Para ver cómo funciona esto, empezar por echar un vistazo a `draw_svg()` *software_renderer.cpp*. El método acepta un archivo SVG y dibuja todos los elementos en el archivo SVG a través de una secuencia de llamadas a `draw_element()`. Para cada elemento `draw_element()` inspecciona el tipo del elemento, y luego llama a la función apropiada. En el caso de puntos, la función es la de `draw_point()`.

La posición de cada punto se define en un marco de coordenadas locales, tal que `draw_point()` primero transforma el punto de entrada a su posición al espacio de pantalla (ver línea `p_screen = transform(p)`). Esta transformación se encuentra al principio de `draw_svg()`. En el código inicial, esta transformación convierte del sistema de coordenadas del canvas de svg a coordenadas de pantalla. Tendrán

que manejar transformaciones más complejas para apoyar más complejos archivos SVG e implementar controles de visualización del mouse más adelante en la asignación.

La función de `rasterize_point()` es responsable de dibujar realmente el punto. En este trabajo definimos el espacio en la pantalla para una imagen salida de tamaño `(target_w, target_h)` como sigue:

- `(0, 0)` corresponde a la parte superior izquierda de la imagen de salida
- `(target_w, target_h)` corresponde a la parte inferior derecha de la imagen de salida
- Asuma que las posiciones de la pantalla están situadas en las coordenadas de mitad-entero en el espacio de pantalla. Es decir, el punto superior izquierdo muestra es en coordenadas (0.5, 0.5), y el punto de muestra abajo a la derecha está en coordenadas (target_w-0.5, target_h-0.5).



Para rasterizar puntos, adoptamos la siguiente regla: un punto abarca a lo más una muestra de la pantalla: la muestra más cercana al punto en el espacio de

pantalla. Esto se implementa como sigue, asumiendo que (x, y) es la ubicación del espacio de pantalla de un punto.

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

Por supuesto, el código no debe intentar modificar el búfer render target en ubicaciones de píxeles no válidas.

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

Si los puntos están en la pantalla, llenamos el pixel con el color RGBA asociado con el punto.

```
  render_target[4 * (sx + sy * target_w)    ] = (uint8_t) (color.r * 255);
  render_target[4 * (sx + sy * target_w) + 1] = (uint8_t) (color.g * 255);
  render_target[4 * (sx + sy * target_w) + 2] = (uint8_t) (color.b * 255);
  render_target[4 * (sx + sy * target_w) + 3] = (uint8_t) (color.a * 255);
```

En este momento el código inicial no controlar correctamente puntos transparentes.

Ahora que entiendes los conceptos básicos de dibujo elementos, pongámonos a trabajar realmente en dibujar elementos más interesantes que los puntos.

Tarea 1: Dibujado de líneas

En esta tarea se agregara la funcionalidad de dibujo de una línea mediante la implementación de la función `rasterize_line()` en*software_renderer.cpp*.

Pueden consultar la web y utilizar cualquier algoritmo que deseen. Sin embargo, su solución debe:

- Manejar coordenadas de vértices no enteros que se dan como argumento a `rasterize_line()`.
- Manejar líneas de cualquier pendiente.

- Realizar trabajo proporcional a la longitud de la línea (los métodos de trabajan para cada píxel en la pantalla o para todas las muestras en el cuadro delimitador de la línea no soluciones aceptables).

Se les anima a comenzar con una implementación del **algoritmo de Bresenham** y después, si lo desea, continuar con las implementaciones que mejoran la calidad (por ejemplo, dibujar líneas suaves) u optimización del rendimiento del dibujado.

Cuando terminen, la solución debe ser capaz de renderizar correctamente *basic/test2.svg*.

Créditos adicionales:

- Si se comparan los resultados iniciales de Bresenham con la implementación de referencia, notarán que la solución de referencia genera líneas suaves. Por ejemplo, podrían modificar su implementación de Bresenham para realizar el algoritmo de línea de Xiaolin Wu_.
- Añadir soporte para especificar un ancho de línea.
- Mejorar el rendimiento: hacer su algoritmo de rasterización de línea muy, muy rápido.

Tarea 2: Elaboración de triángulos

En esta tarea, se implementará `rasterize_triangle()` en *software_renderer.cpp*.

Su implementación debe:

- El relleno de un triángulo, mientras que en la tarea 1 se da elección en cómo se definen las salidas de línea de dibujo, hay una solución exacta al problema del relleno de un triángulo. La posición de la pantalla de puntos

de muestra - en coordenadas de mitad-entero en el espacio de la pantalla- fue descrita anteriormente.

- Para recibir el crédito completo en la tarea 2 su implementación debe asumir que un punto de muestra en un extremo del triángulo está cubierto por el triángulo. Su implementación **no tiene que respetar** las "reglas de borde" del triángulo para evitar la "doble contabilidad" como se explica en clase.

- Su aplicación debe utilizar un algoritmo que sea más eficiente que simplemente probar todas las muestras en pantalla. Para recibir el crédito completo deben restringir al menos las pruebas de cobertura para las muestras que se encuentran dentro de un cuadro delimitador de espacio de pantalla del triángulo. Sin embargo, se les anima a explorar implementaciones más eficientes.

- Cuando un triángulo cubre una muestra, se debe escribir el color del de triángulo de acuerdo a la ubicación correspondiente a esta muestra en `render_target`.

Una vez que se ha implementado con éxito el dibujo del triángulo, podrás dibujar un gran número de ejemplos. Al cargar un SVG, el código proporcionado triangula poliedros convexos en una lista de triángulos, así que implementar el soporte para rasterizar triángulos, el visor de dibujo soporta cualquiera de estas formas también. (Al analizar el SVG, convertimos rectángulos y polígonos especificados en el archivo en las listas de los triángulos).

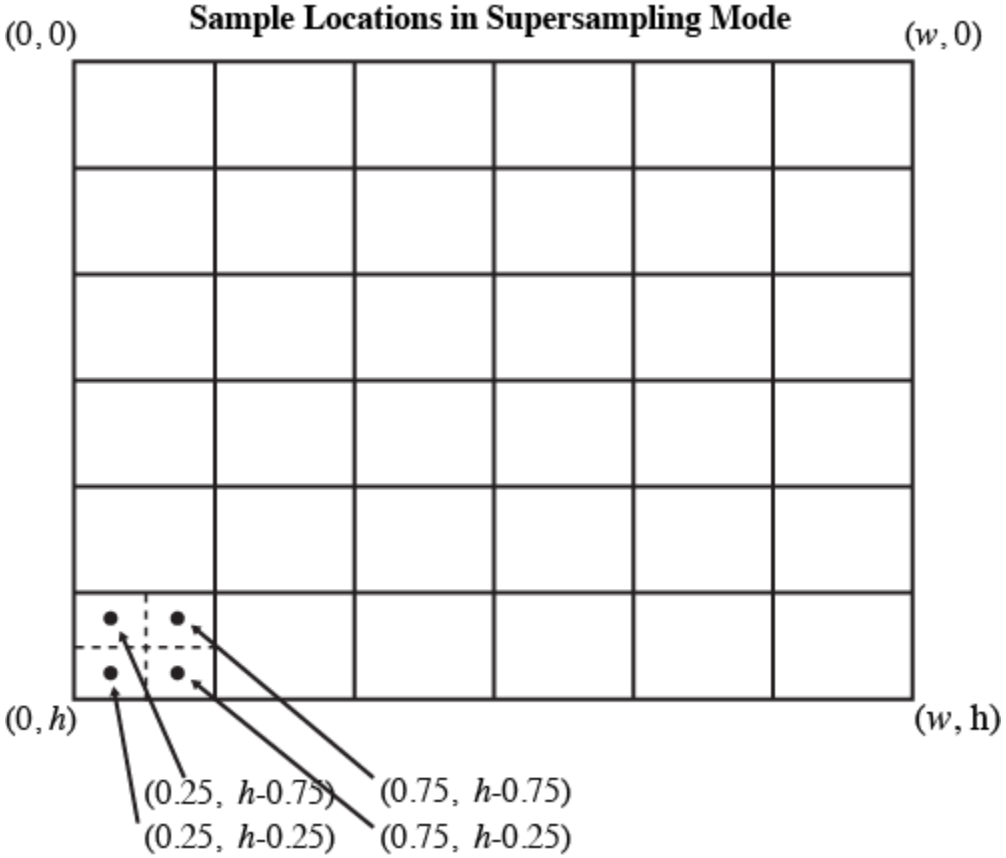
Cuando haya terminado, podrá llamar *basic/test3.svg*, basic/test4.svg y basic/test5.svg.

Extensiones de crédito adicional posible:

- Mejorar el rendimiento: hacer el rasterizador de triángulo muy, muy rápido.

Tarea 3: Uso de Anti-Aliasing Supersampling

En esta tarea, se extenderá el rasterizador para suavizar de aristas de triángulos través de supersampling. En respuesta a que el usuario cambie la velocidad de muestreo de la pantalla (las teclas `=` y `-`), la aplicación llamara `set_sample_rate()`. El parámetro `sample_rate` define la frecuencia de muestreo de cada dimensión, así que para un valor de 2 corresponde a una densidad de muestra de 4 muestras por píxel. En este caso, las muestras dentro del píxel superior izquierdo de la pantalla se ubicarían en las posiciones (0.25, 0.25) (0,25, 0,75), (0,25, 0,75) y (0.75, 0.75).



Es razonable pensar en el rendering supersampled como el rendering de una imagen que es `sample_rate` veces más grande que la imagen de salida real en cada dimensión, entonces, volver a muestrear el resultado representado más grande hasta el muestreo de pantalla después de completar el rendering. Aquí hay un boceto de una implementación. **Nota: Si implementan el rasterizador de triángulo en términos de muestreo de cobertura en coordenadas del espacio de pantalla (y no en términos de pixeles), entonces los cambios de código para apoyar supersampling deberían ser bastante simples para los triángulos:**

- Al rasterizar primitivas tales como triángulos, en lugar de actualizar directamente `render_target`, la rasterización debe actualizar el contenido de un buffer más grande (quizás llamarlo `supersample_target`) que contiene los resultados de la muestra de súper. Ustedes tendrán que asignar/liberar este buffer.

- Después de finalizada la representación, su implementación debe volver a muestrear el búfer de resultados “supersampled” para obtener los valores de la muestra para el render target. Esto a menudo se llama "resolver" el supersampled búfer en el render target. Implementar el re muestreo utilizando un filtro de caja simple de área unitaria.

  Tengan en cuenta que la función `SoftwareRendererImp::resolve()` es llamada por `draw_svg()` después de que el archivo SVG se ha dibujado. Por lo tanto, es muy conveniente para llevar a cabo el re muestreo aquí.

- Puede que necesite tomar medidas adicionales en `SoftwareRendererImp::clear_target()` en modo supersampling.

Cuando terminen, prueben incrementar el supersampling en el viewer.

También observen que después de habilitar el procesamiento supersampled, algo podría haber ido muy mal con la rasterización de líneas y puntos. (Pista: probablemente parecen más delgados). Modifiquen la implementación de líneas y puntos de rasterización para que la representación “supersampled” de estas primitivas conserva su espesor a través de diferentes tipos de supersampling. (Una solución que no realiza antialiasing en líneas y puntos es aceptable).

Posible crédito adicional de extensiones:

- Aplicar anti-aliasing morfológico (MLAA), en lugar de supersampling. Es sorprendente cómo funciona esto. MLAA es una técnica utilizada en toda la comunidad de juegos para evitar el alto costo de supersampling, pero todavía evita los artefactos de imagen desagradable causados por el aliasing. (Una versión más avanzada de MLAA está aquí).
- Implementar jitter de muestreo para mejorar la calidad de imagen cuando se esté en supersampling.
- Implementar filtros de re muestreo de mejor calidad que el de un cuadro y analizar su impacto en la calidad de la imagen.

Tarea 4: Aplicación de modelado y visualización de transformaciones

Parte 1: Modelado de transforma

En esta tarea se extenderá el renderizador para interpretar adecuadamente la jerarquía de las transformaciones de modelación en los archivos SVG.

Hay que recordar que un objeto SVG consiste en una jerarquía de elementos de forma. Cada elemento en un SVG está asociado a una transformación de modelado (ver `SVGElement.transform` en *svg.h*) que define la relación entre el espacio de coordenadas local del objeto y el espacio de coordenadas del

elemento padre. En realidad, la implementación de `draw_element()` ignora estas transformaciones de modelado, así que los únicos objetos SVG que su procesador ha sido capaz de establecer correctamente eran objetos que contenían sólo transformaciones de modelado identidad.

Modificar `draw_svg()` y `draw_element()` aplicar la jerarquía de transformaciones especificada en el objeto SVG. (Sí, esto es una tarea sencilla que debería incluir no más de unas pocas líneas de código.)

Cuando haya terminado, deberan poder llamar *basic/test6.svg*.

Parte 2: Visión transformar

Para implementar esta funcionalidad en su solución, tendrán que implementar `ViewportImp::set_viewbox()` en `viewport.cpp`.

Un viewport define una región del canvas de SVG que es visible en la aplicación. Cuando la aplicación se ejecuta inicialmente, el canvas entero está a la vista. Por ejemplo, si el canvas SVG es de tamaño 400 x 300, entonces la vista estará centrada inicialmente en el centro del canvas y tiene un campo de visión vertical que se extiende al canvas entero. Específicamente, los valores de la clase `viewport` serán: `x = 200, y = 150, span = 150`.

Cuando las acciones del usuario requieren cambiar el viewport, la aplicación llamará a `update_viewbox()` con los parámetros adecuados. Dado este cambio en parámetros de vista, se debe implementar `set_viewbox()` para calcular una transformación `canvas_to_norm` basada en los nuevos parámetros de vista. Esta transformación debe asignar el espacio de coordenadas del canvas SVG a un espacio normalizado donde la parte superior izquierda del viewport se mapea a (0,0) y la parte inferior derecha se mapea a (1, 1). Por ejemplo, para los valores `x = 200, y = 150, span = 10`, entonces las coordenadas SVG (190, 140) se

transforman a las coordenadas normalizadas (0, 0) del canvas y coordenadas de canvas (210, 160) se transforman a (1, 1).

Una vez han implementado correctamente `set_viewbox()`, su solución responderá a los controles del mouse en la misma forma que la implementación de referencia.

Tarea 5: Imágenes

En esta tarea, se implementará `rasterize_image()` en *software_renderer.cpp*.

Se va a limitar este problema a rasterizar elementos imagen que se colocan en canvas de SVG a través de las traslaciones y escalamientos, **pero no las rotaciones**. Por lo tanto,`rasterize_image()` debe renderizar la imagen especificada en un rectángulo alineado al eje en la pantalla cuya coordenada superior son `(x0, y0)` y cuya coordenada inferior derecha es `(x1, y1)`. Su implementación debe cumplir las siguientes especificaciones:

- El elemento de la imagen debe cubrir todas las muestras de la pantalla dentro del rectángulo especificado.
- La asignación del espacio de pantalla al espacio de textura es el siguiente: `(x0, y0)` en el espacio de pantalla se mapea a la coordenada de textura de la imagen a `(0, 0)` y `(x1, y1)` a `(1, 1)`.
- Pueden mirar la implementación de imágenes de textura de entrada en `Texture.h/.cpp`. La clase `Sampler2D` proporciona el esqueleto de los métodos para **el vecino más cercano** (`sampler_nearest()`), **filtrado bilineal** (`sampler_bilinear()`) y el **trilineal** (`sample_trilinear()`). En esta tarea, el color de la imagen en la ubicación especificada de la muestra se calculará con **filtrado bilineal** de la textura de la entrada. Por lo tanto se debe implementar `Sampler2D::sampler_bilinear()` en `texture.cpp` y llamarse

desde `rasterize_image()`. (Sin embargo, se recomienda primero aplicar `Sampler2D::sampler_nearest()` – ya que el filtrado del vecino más es más simple y se dará crédito parcial.)

- Asuman que los píxeles de la imagen corresponden a muestras en coordenadas mitad-entero en el espacio de textura.
- La estructura `Texture` que se almacena en la clase `Sampler2D` mantiene varios búferes de imagen que corresponden a una jerarquía de mipmap. En esta tarea, se muestrea desde el nivel 0 de la jerarquía: `Texture::mipmap [0]`.

Cuando hayan terminado, deberan poder llamar *basic/test7.svg*.

Tarea 6: Elementos imagen: Anti-Aliasing usando filtrado trilineal

En esta tarea mejorará su suavizado de elementos de imagen mediante la adición de **filtrado trilineal**. Esto implicará generar mipmaps para los elementos de imagen en tiempo de carga de SVG y después modificar su código de la tarea 5 para implementar un **filtrado trilineal** usando el mipmap. Su implementación sólo necesita trabajar para imágenes que tienen dimensiones en cada dirección base 2.

- Para generar mipmaps, necesita modificar código en `Sampler2DImp::generate_mips()` en *texture.cpp*. Se da el código para asignar todos los buffers adecuados para cada nivel de la jerarquía del mipmap. Sin embargo, tendrá que llenar el contenido de estos buffers con los datos originales de la textura en el nivel 0. **Su aplicación puede asumir que todas las imágenes de textura de entrada tienen dimensión base dos. (Pero no debe asumir las imágenes entradas son cuadradas)**

- Luego modificar la implementación de `rasterize_image()` de la tarea 5 para realizar el muestreo del filtrado trilineal del mipmap. Su implementación deberá primero calcular el nivel apropiado del cual se muestrea de la jerarquía de mipmap. Los elementos imagen se encojen en la pantalla, para evitar el aliasing el rasterizador debe muestrear de niveles más altos de la jerarquía del mipmap.

En este punto, hacer zoom dentro y fuera de la imagen debe producir resultados con un buen bien filtrado

Tarea 7: Composición alfa

Hasta este punto, su procesador no puede dibujar correctamente elementos semi-transparentes. Por lo tanto, su última tarea de programación en esta tarea es modificar el código para implementar el Simple Alpha Blending en la especificación de SVG.

Mientras que la aplicación siempre borrará el búfer de render objetivo al color del canvas al comienzo de un frame a un blanco opaco (RGBA en (255,255,255,255)) antes de dibujar cualquier elemento SVG, la implementación de la transparencia no debe hacer ninguna hipótesis sobre el estado del objetivo al principio de un frame.

Cuando hayan terminado, deberán poder establecer correctamente las pruebas en */alpha*.

Tarea 8: Dibujar algo!

Ahora que han implementado algunas características básicas del formato SVG, es tiempo de ser creativo y dibujar algo. Pueden crear un archivo SVG con herramientas de diseño populares **como Adobe Illustrator** o **Inkscape** y exportar los archivos al formato SVG. Sin embargo, tenga en cuenta que nuestro código de arranque y la implementación de procesador sólo admiten un

subconjunto de las características definidas en la especificación de SVG, y estas aplicaciones no siempre pueden codificar las formas con las primitivas que realizamos. (Puede que necesite convertir caminos complicados a las primitivas básicas de estas herramientas). **También, no es muy difícil escribir archivos SVG directamente ya que son simplemente archivos XML.**

Yendo más lejos: Las tareas que pueden potencialmente ganar usted Extra crédito:

Implementar formas más avanzadas

Se proveen un par de ejemplos de subdividir formas complejas, suaviza formas en triángulos mucho más simples en */subdiv*. Pueden ver subdivisión en acción, en los archivos de prueba que se proporcionan.

Además de lo que han implementado ya, las Formas básicas de SVG también incluyen círculos y elipses. Podemos soportar estas características convirtiéndolas a polígonos triangulados. Pero si nos acercamos a los bordes, habrá un punto en que la aproximación se descompone y la imagen no se verá como una curva suave. Un triangulando más fino puede ser costoso ya que un gran número de triángulos puede que sea necesario para obtener una buena aproximación. ¿Hay una mejor manera de probar estas formas? Por ejemplo, implementar `drawEllipse` en *drawsvg.cpp*

Mejorar el rendimiento

Lograr el alto rendimiento es crítico en muchas aplicaciones gráficas. Se está muy interesado en quien puede generar el proceso más rápido en la clase.

Calificación

Cada tarea se calificará sobre la base de corrección. No se espera reproducir totalmente la solución píxel por píxel ya que ligeras diferencias en la estrategia de ejecución o incluso el orden de la aritmética de punto de flotante genera diferencias, pero su solución no estar muy lejos.

El proyecto se divide en dos entregas:

- **En su primera entrega deberán realizar las tareas de la 1 a la 4.**
- **En la segunda entrega deberán realizar las tareas de la 5 a la 8.**

La asignación consta de un total de 100 pts. Los puntos son los siguientes:

- Tarea 1: 5
- Tarea 2: 20
- Tarea 3: 20
- Tarea 4: 10
- Tarea 5: 15
- Tarea 6: 20
- Tarea 7: 5
- Tarea 8: 5

ANEXO

FreeType

La biblioteca Freetype es necesaria para poder compilar su proyecto. Esta biblioteca la pueden descargar de la página http://www.freetype.org/.

Instalación de FreeType

La instalación de Freetype es muy sencilla. Descompriman el archivo freetype-2.X.X. tar.gz, una vez hecho esto abrir una terminal y entrar al directorio fuente

..\..\..\freetype-2.X.X

Una vez dentro se realiza los siguiente:

- Escriban ./configure.
- Una vez que termine, escriban make.
- Finalmente escriban make install.

Con esto habrán terminado de compilar e instalar la biblioteca de FreeType.