Computación Gráfica - TP: Texture Painting

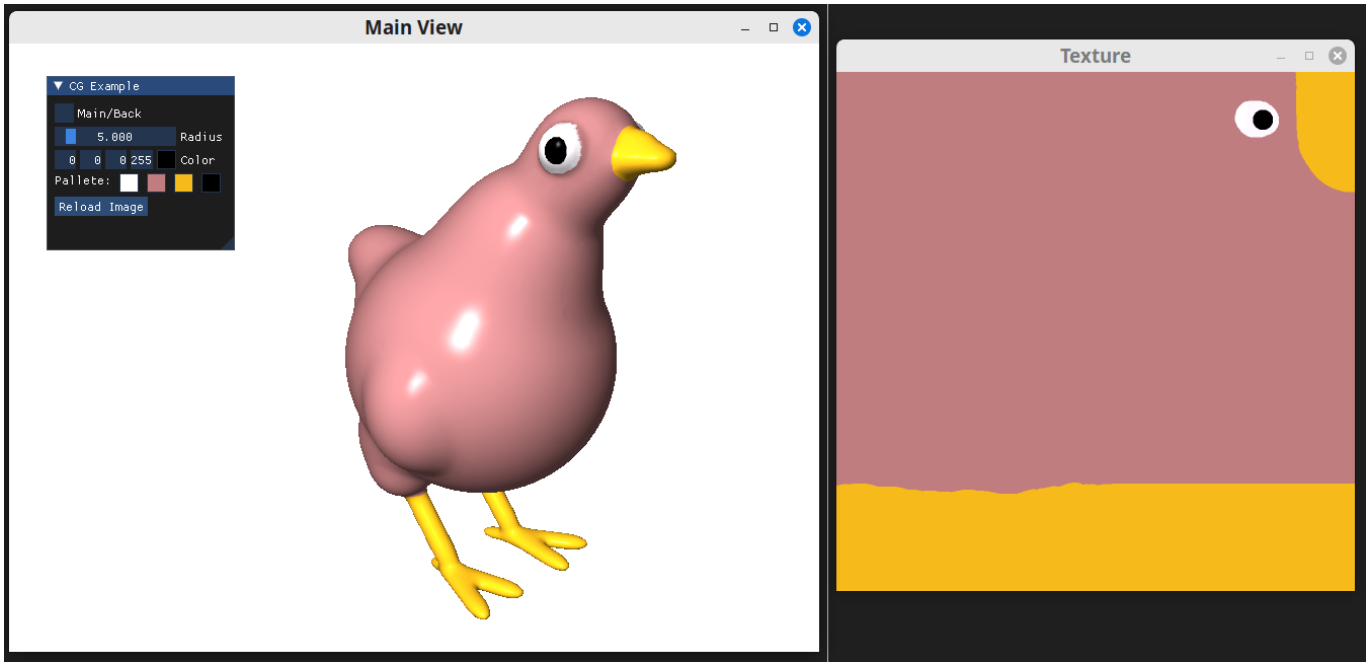
1. Resumen de tareas

El objetivo del TP es permitir al usuario pintar la textura directamente sobre el objeto 3D (como el pincel de un paint, pero sobre el modelo, no sobre la imagen de la textura). Para ello deberá:

1. Implementar algoritmos de rasterización para dibujar a mano alzada sobre la imagen que se utiliza como textura.
2. Implementar un mecanismo image-presition para obtener, a partir de un click en la vista 3D, las coordenadas de textura correspondientes.

2. Consigna detallada

El práctico crea dos ventanas: una ventana principal con la vista 3D habitual del modelo (*main*), y una ventana auxiliar (*aux*) donde se muestra la textura que el modelo utiliza. Luego de resuelto el práctico, el usuario debería poder ”pintar” con el ratón en cualquiera de las dos para modificar la textura. En la primera parte del TP, deberá resolver el problema de ”pintar” sobre la ventana auxiliar, ya que allí las coordenadas del ratón se traducen de forma muy simple a coordenadas en la imagen, y entonces el problema consistirá solamente en adecuar los callbacks e implementar el algoritmo de rasterización. En la segunda parte deberá resolver el problema de pintar en la ventana principal, y allí deberá implementar una técnica image-presition para convertir las coordenadas de los clicks y movimientos del mouse en la vista 3D a coordenadas 2D sobre la imagen de la textura.



Las funciones a completar en el código tienen un comentario `/// @TO-DO: .....` En principio no es necesario modificar ninguna otra función de las preexistentes (pero puede agregar todas las funciones auxiliares que quiera). Es útil también mirar las variables globales disponibles (al comienzo de *main.cpp*, y también puede agregar allí las que necesite). No debería ser necesario modificar nada fuera del archivo *main.cpp* y del shader que usted cree para la segunda parte. Sí es conveniente analizar y entender mínimamente *main.cpp* completo, pero no debería ser

necesario abrir ninguna de las clases o funciones auxiliares (entendiendo el *pipeline* debería ser relativamente obvio para qué sirven).

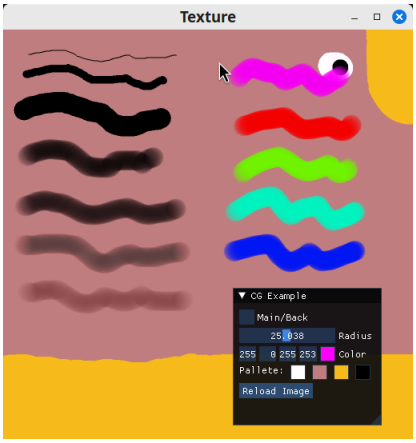
Verá que ambas ventanas tienen definidos los callbacks para click y motion del mouse. En la primera parte deberá trabajar solo con los callbacks de la ventana auxiliar. En la segunda parte deberá modificar los de la ventana principal. Si en un callback no dispone de las coordenadas del mouse, para obtenerlas puede utilizar `glfwGetCursorPos(window, &xpos, &ypos);` siendo `xpos` e `ypos` variables de tipo `double`. Si quiere saber los límites para estas coordenadas (el tamaño del viewport en la ventana), puede utilizar `BufferSize bs = getBufferSize(window);`, donde `bs` será un struct con los campos `width` y `height`.

2.1. Dibujando sobre la textura

El primer paso será entonces lograr dibujar con el ratón en la ventana auxiliar (la que solo muestra la textura). Si se hace un click simple se debe dibujar solo un punto. Si se mueve el ratón con el botón izquierdo apretado se debe generar una curva contigua. Desde los controles agregados con *ImGui* en la ventana principal, se debe poder controlar el ancho y color de la curva.

La instancia `image` de la clase `Image` representa a la imagen que se usa de textura, cargada en memoria RAM para poder editarla desde la aplicación. Con el método `SetRGB` de esa clase puede modificar los píxeles de esa imagen (que serán en realidad los téxeles). `SetRGB` que actualiza un texel: recibe fila, columna, y valores entre 0 y 1 para los canales R, G y B. `Image` tiene además los métodos `GetWidth()` y `GetHeigth()` por si necesita verificar si un índice de fila o columna está en el rango correcto.

En los algoritmos de rasterización debe usar las variables globales `radius` (radio en téxeles del círculo a pintar cuando es un punto; o semi-ancho de la curva/segmento cuando se mueve el mouse) y `color` (color con el que pintar, incluyendo un canal *alpha* que controla la opacidad).



Ejemplo de trazos de diferentes radios/colores/opacidad.

Sugerencia: primero intente rasterizar correctamente el trayecto del ratón con una linea/curva de 1 píxel de ancho y sin considerar el canal *alpha*; luego modifique el algoritmo para considerar el radio; y finalmente intente considerar también el canal *alpha* del color.

No se espera que los algoritmos de rasterización que implemente sean súper-eficientes, solo ”razonablemente” eficientes (cuidar el orden, no la constante). Sí se espera que sean correctos. Esto quiere decir que no es aceptable si no se respeta, por ejemplo, la contigüidad; pero sí es perdonable si pinta más de una vez un mismo píxel (siempre que el repintado esté acotado).

Luego de modificar la imagen (lo cual ocurre en RAM), deberá incluir la linea `texture.update(image)` para transferir los cambios a la GPU para que la modificación se ”vea” (de lo contrario, la GPU seguirá utilizando su copia de la imagen original).

2.2. De coords del click a coords de textura

Si ya implementó los mecanismos de rasterización para la ventana 2D, ahora debará reutilizarlos desde la ventanan 3D. La idea es poder pintar directamente allí, pero para lograrlo será necesario traducir los clicks en la vista 3D a coordenadas en la textura (*s,t*, o fila/columna en la matriz de la imagen). Para ello utilizaremos una técnica que lo resuelve en el espacio de la imagen (en el mundo ráster, en contraste con resolverlo geométricamente en el modelo vectorial). La idea es renderizar el objeto pintándolo con las coordenadas de textura, para luego averigua el color debajo del píxel donde el usuario haga click y a partir del mismo determinar las coordenadas de textura.

2.2.1. ”Pintar” las coordenadas de textura

El primer paso será entonces lograr ”pintarlo” con las coordenadas de textura. Para ello deberá completar la función `drawBack` para que dibuje al modelo con un nuevo shader especial que aplique los valores de las coordenadas de textura directamente como color. Puede basarse en el código de la función `drawMain`, que es la que dibuja lo que efectivamente se ve, para plantear el código de `drawBack`, y tomar como ejemplo los shaders que trae el TP para plantear el nuevo.

Sugerencia: comience reemplazando en el loop principal `drawMain` por `drawBack` para ver el efecto de la función y el shader que implemente. Pero luego, este renderizado auxiliar para obtener las coordenadas de textura, deberá ser invisible al usuario. Agregue a la GUI (con ImGUI) un checkbox para optar por visualizar ese renderizado alternativo en lugar del normal.

2.2.2. Dibujar sobre la vista 3D

Una vez logrado esto, modifique los callbacks del mouse de la ventana principal para invocar a `drawBack`, obtener las coordenadas de textura del ”pixel” donde está el ratón, y pasarle esos datos a los algoritmos de rasterización 2D que ya resolvió en la primera mitad del TP (reutilice código, no vuelva a implementar DDA!). Notar que además de determinar las coordenadas de textura para un fragmento del modelo sobre el que se hace click, primero deberá pensar cómo determinar si efectivamente se hizo click sobre el modelo o sobre el fondo.

Ayuda: para leer un valor de una posición de un buffer, se utiliza la función `glReadPixels`, que puede leer todo un rectángulo y desde cualquiera de los buffers (con `glReadBuffer` decimos previamente cuál). Por ejemplo:

```
// leer el valor de profundidad del depth buffer para el fragmento x,y
glReadBuffer(GL_DEPTH);
float depth_value;
glReadPixels(x,y,1,1, GL_DEPTH_COMPONENT, GL_FLOAT, &depth_value);

// leer el valor de color del back-buffer para el fragmento x,y
glReadBuffer(GL_BACK);
glm::vec4 color_value;
glReadPixels(x,y,1,1, GL_RGBA, GL_FLOAT, &(color_value[0]));
```

En los ejemplos, la posición del fragmento a leer está dado por *x,y*, siendo *0,0* el primer fragmento de la imagen/buffer (arriba a la derecha), y *ancho-1,alto-1* el último (abajo a la izquierda). Los resultados (variables `depth_value` o los canales de `color_value`) serán valores reales entre 0 y 1.

2.2.3. Posibles problemas

Finalmente, pruebe dibujar lentamente una linea de 1px de ancho. Es probable que la curva rasterizada desde la vista 3D se vea escalonada, y no sea culpa del pixelado de la textura ni de defectos del algoritmo de rasterización, sino de problemas de precisión de su implementación de esta técnica, que deberá entender y corregir. En la imagen de la izquierda, el serruchado excesivo es producto del método para determinar la coordenadas de textura (y se verá también en la vista 2D); mientras que a la derecha está corregido y el serruchado que queda sí se corresponde con la resolución de la textura.