Universidad Nacional
Autónoma de México

Facultad de Ingeniería

Ingeniería en Computación

Práctica 3
Terrenos

Alumno(a)
Isabel Gómez Yareli Elizabeth
Ortiz Figueroa María Fernanda
Roldan Rivera Luis Ricardo

Asignatura
Computación Gráfica Avanzada

Grupo
1

Profesor
M.C. Reynaldo Martell Ávila

Fecha de entrega
27 de febrero de 2020

2020 - 2

Objetivo

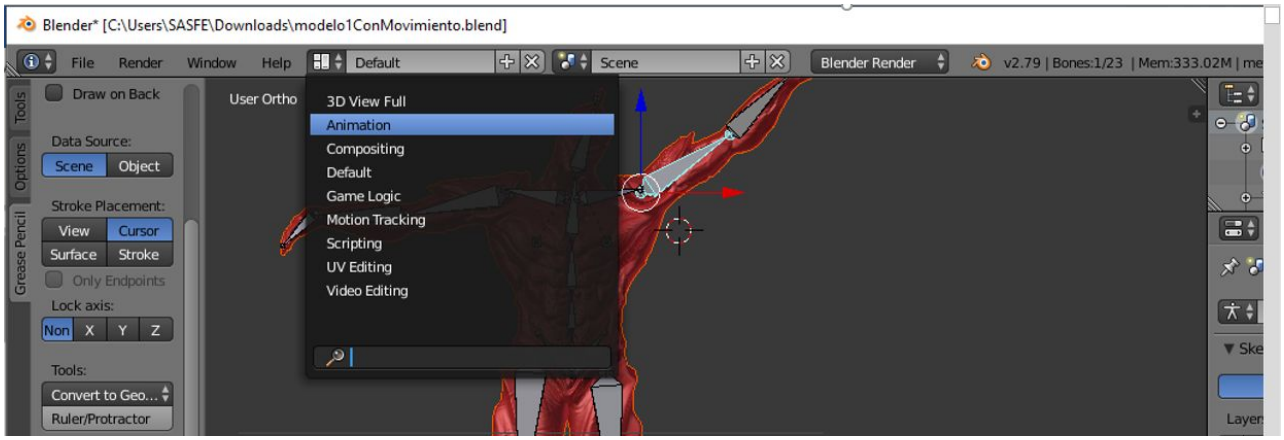
Crear un modelo 3D con múltiples animaciones en Blender, exportar este en formato *.fbx y controlar las animaciones por medio de alguna tecla en un proyecto en Visual Studio con OpenGL.

Crear e implementar una imagen en escala de grises para crear un terreno en un proyecto de Visual Studio con OpenGL para poder visualizar terrenos con diferentes alturas.

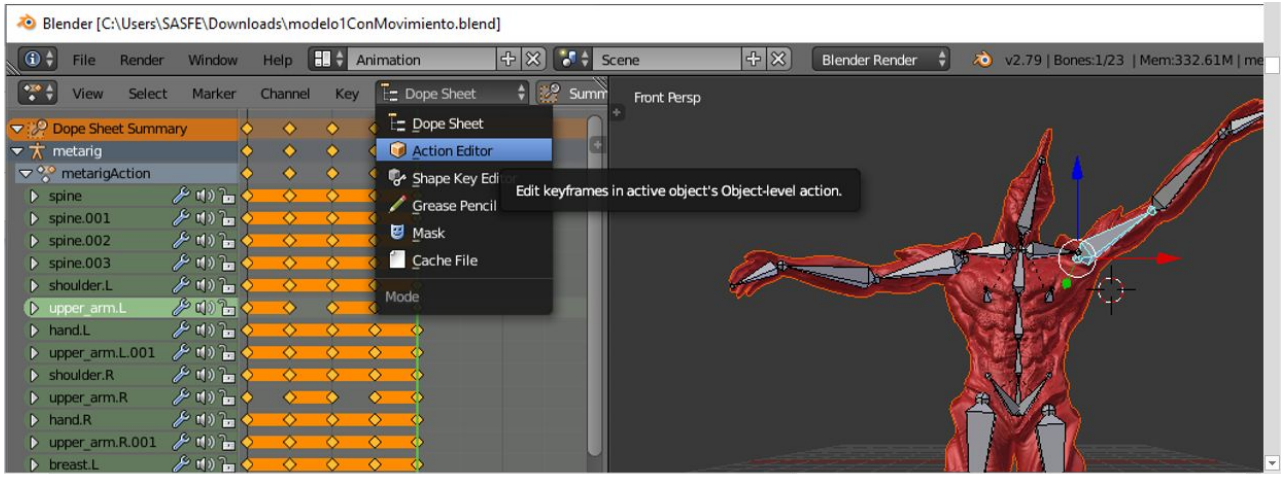
Desarrollo

Animaciones múltiples en un modelo 3D

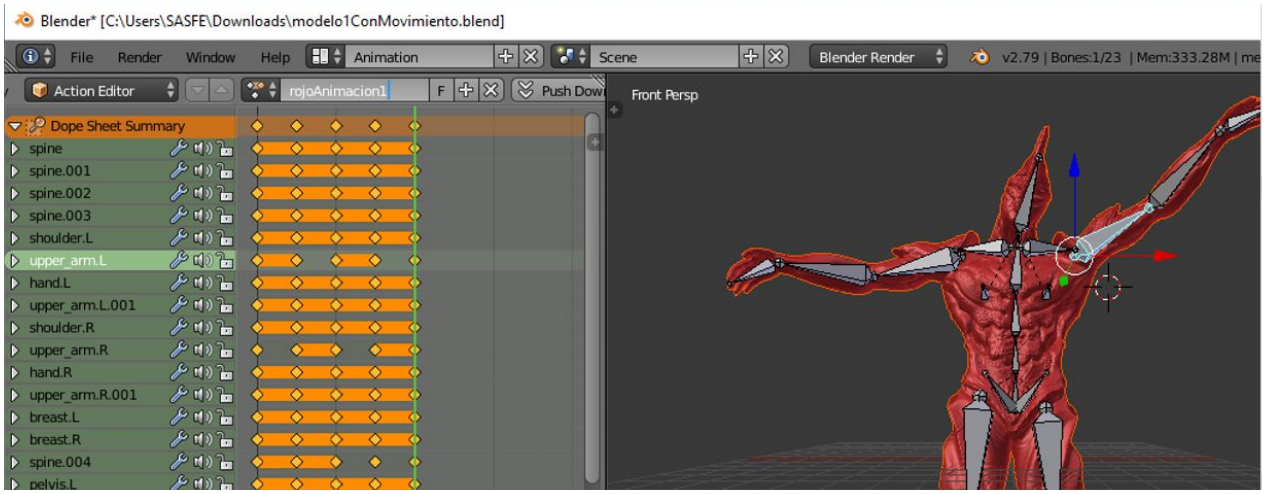
En la práctica anterior se realizó la animación de un modelo en diferentes tiempos a partir de cuadros clave, partiendo de este proyecto, en caso de requerir que exista más de una animación, procedemos a generar una nueva animación, para lo cual debemos asegurarnos de encontrarnos en modo Animation.



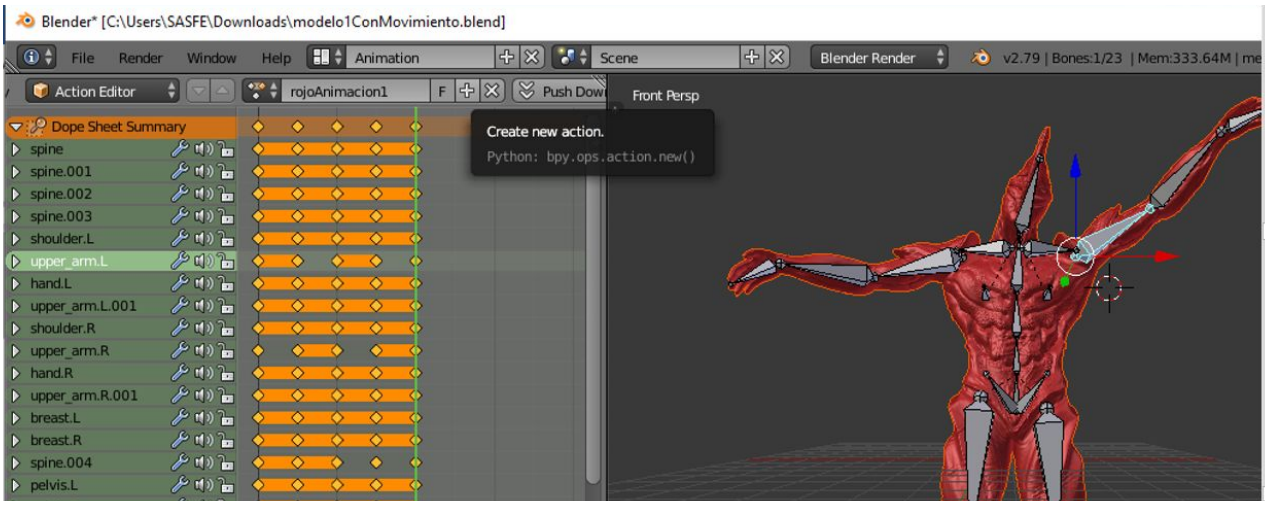
Después, seleccionamos en el panel izquierdo de la pantalla en la barra de herramientas el modo *Action Editor*.



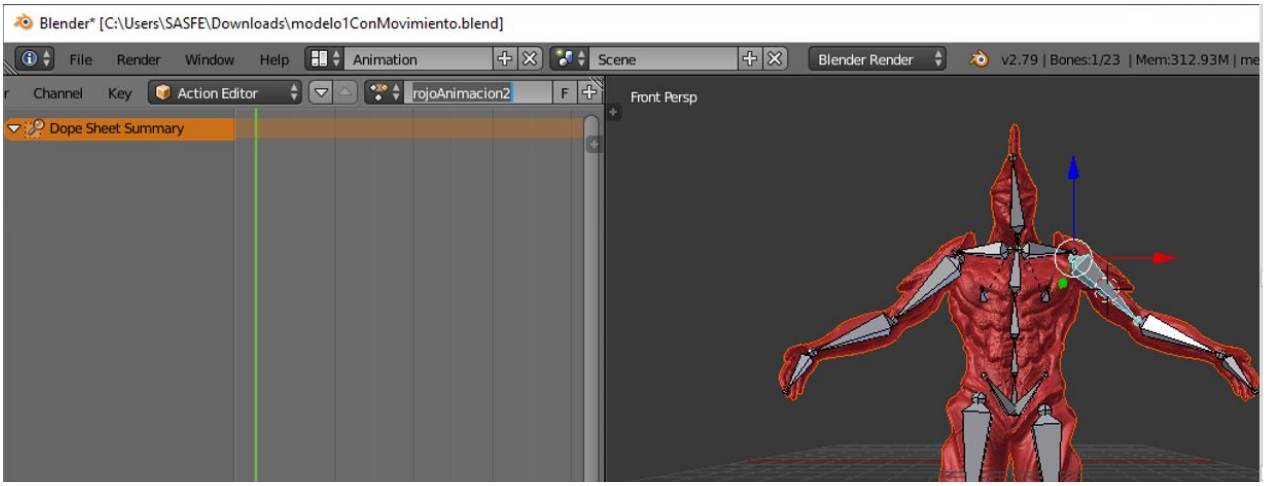
Para llevar un orden en las animaciones que vamos creando, asignamos algún nombre que nos permita identificarlas, primero a la animación que ya teníamos.



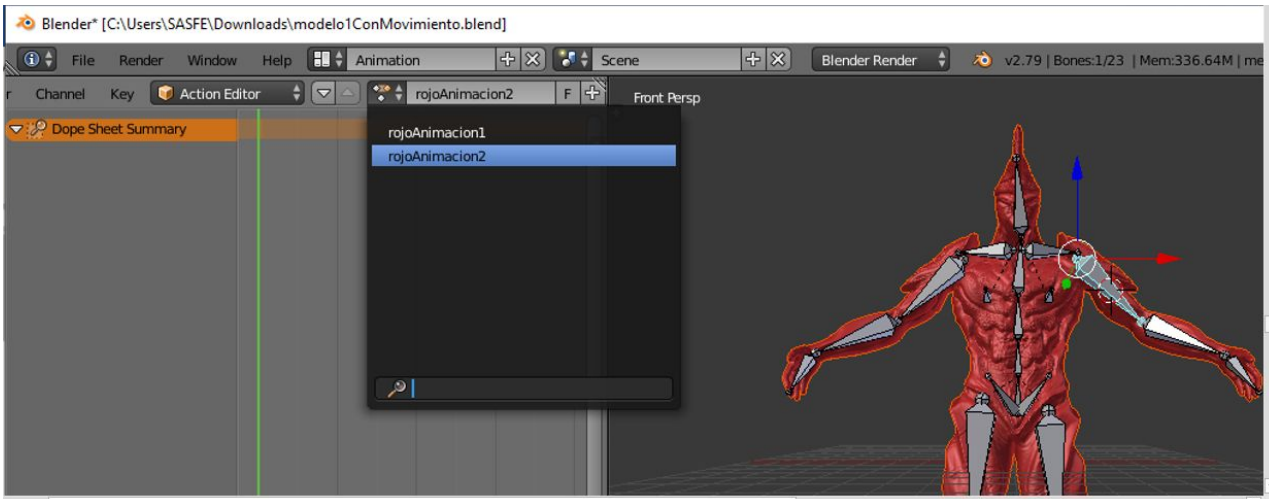
Luego, damos click al botón con el símbolo de +, el cual corresponde a *Create a new action*, en caso de no aparecer de esta manera identificar un botón que diga *New*, ambos tienen la misma función.



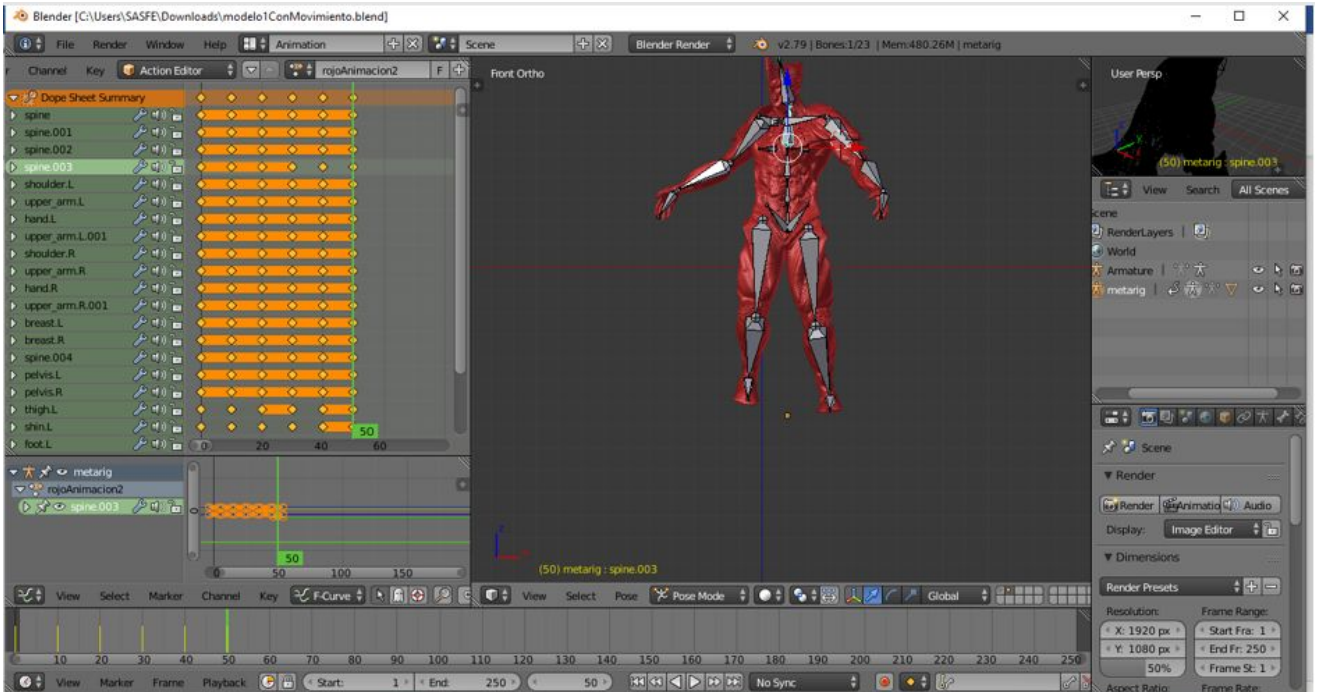
Como en la animación que ya teníamos, le asignamos un nombre a la que ha sido creada.



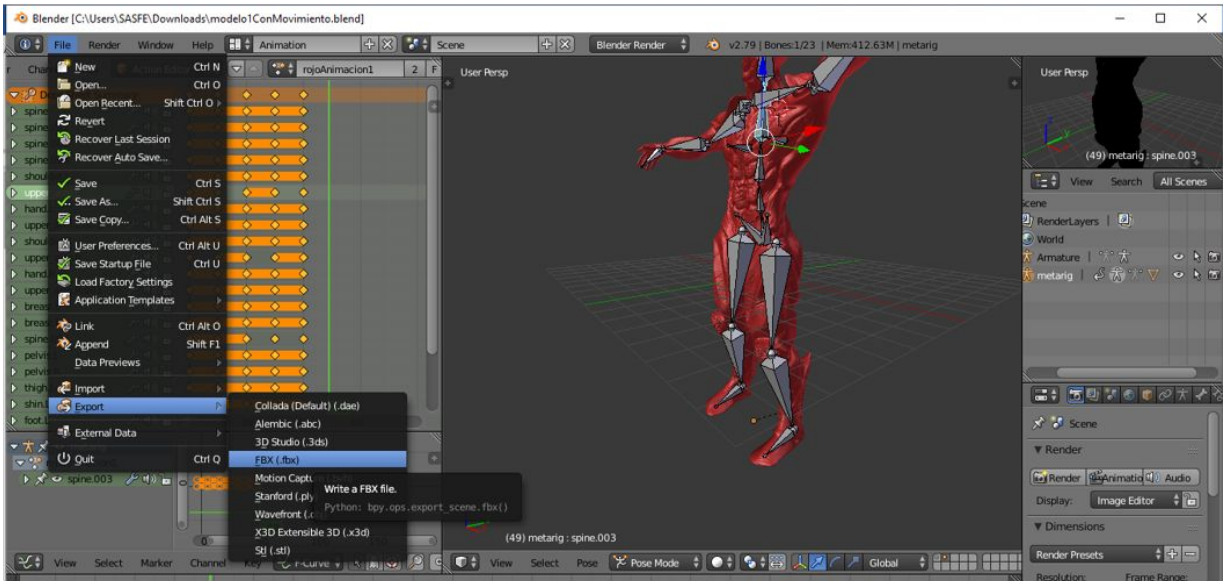
Llegados a este punto, podemos observar que ahora contamos con un modelo con dos tiempos diferentes de animaciones.



Posteriormente, seleccionamos el esqueleto del modelo, nos cambiamos a *Pose Mode*, y procedemos a generar cada unos de los cuadros claves a aplicar diversas transformaciones en diferentes tiempos, al presionar la tecla I y elegir la opción *Whole Character*, esto se hace hasta crear todos los movimientos correspondientes a dicha animación.



Finalmente, exportamos el modelo en formato *.fbx, para poder implementarlo en el proyecto de la presente práctica.



Implementación de un modelo 3D con múltiples animaciones

Para poder implementar un modelo 3D con múltiples animaciones, lo primero que debemos hacer es agregar el archivo *.fbx generado desde Blender a la respectiva carpeta de *Models* en nuestro proyecto, así como las diversas instrucciones para agregar un nuevo modelo en el *main* de nuestra práctica en Visual Studio con OpenGL, las cuales se muestran a continuación, es importante mencionar que también se muestra tanto como agregar el modelo de Cowboy (el que se agregó y explicó en clase), y el modelo Rojo (el que creamos con las animaciones).

```
81  // Model animate instance
82  // Mayow
83  Model mayowModelAnimate;
84  // Cowboy
85  Model modelAnimateCowboy;
86  // Rojo
87  Model rojoModelAnimate;
```

```
125 // Model matrix definitions
126 glm::mat4 matrixModelRock = glm::mat4(1.0);
127 glm::mat4 modelMatrixHeli = glm::mat4(1.0f);
128 glm::mat4 modelMatrixLambo = glm::mat4(1.0);
129 glm::mat4 modelMatrixAircraft = glm::mat4(1.0);
130 glm::mat4 modelMatrixDart = glm::mat4(1.0f);
131 glm::mat4 modelMatrixMayow = glm::mat4(1.0f);
132 glm::mat4 modelMatrixRojo = glm::mat4(1.0f);
133 glm::mat4 modelMatrixCowboy = glm::mat4(1.0f);
```

```
296     // Mayow
297     mayowModelAnimate.loadModel("../models/mayow/personaje2.fbx");
298     mayowModelAnimate.setShader(&shaderMulLighting);
299
300     // Cowboy
301     modelAnimateCowboy.loadModel("../models/cowboy/Character Running.fbx");
302     modelAnimateCowboy.setShader(&shaderMulLighting);
303
304     // Rojo
305     // El archivo rojoAnimado1 contiene solo una animación
306     // El archivo rojoAnimado2 contiene solo dos animaciones
307     rojoModelAnimate.loadModel("../models/rojo/rojoAnimado2.fbx");
308     rojoModelAnimate.setShader(&shaderMulLighting);
```

```
541 // Custom objects animate
542 mayowModelAnimate.destroy();
543 modelAnimateCowboy.destroy();
544 rojoModelAnimate.destroy();
```

```
735     modelMatrixMayow = glm::translate(modelMatrixMayow, glm::vec3(13.0f, 0.05f, -5.0f));
736     modelMatrixMayow = glm::rotate(modelMatrixMayow, glm::radians(-90.0f), glm::vec3(0, 1, 0));
737
738     modelMatrixCowboy = glm::translate(modelMatrixCowboy, glm::vec3(1.0f, 1.0f, 3.0f));
739
740     modelMatrixRojo = glm::translate(modelMatrixRojo, glm::vec3(6.0f, 1.0f, -5.0f));
741     modelMatrixRojo = glm::rotate(modelMatrixRojo, glm::radians(-90.0f), glm::vec3(0, 1, 0));
```

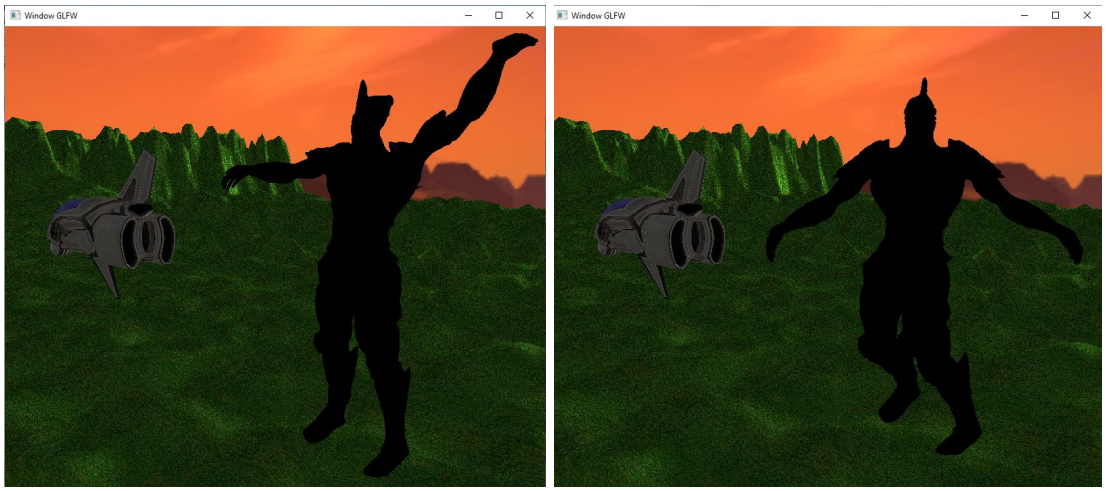
Cabe destacar, que el método *setAnimationIndex*, es el que controla qué animación visualizar, por lo que solo hay que especificarle cual de las animaciones queremos ver, en este caso solo se tienen dos, por lo que bastará con usar una variable de tipo entera para indicar dicho número.

```
925         /*******************************************
926          * Custom Anim objects obj
927          *******************************************/
928         modelMatrixMayow[3][1] = terrain.getHeightTerrain(modelMatrixMayow[3][0], modelMatrixMayow[3][2]);
929         glm::mat4 modelMatrixMayowBody = glm::mat4(modelMatrixMayow);
930         modelMatrixMayowBody = glm::scale(modelMatrixMayowBody, glm::vec3(0.021, 0.021, 0.021));
931         mayowModelAnimate.setAnimationIndex(0);
932         mayowModelAnimate.render(modelMatrixMayowBody);
933
934         modelMatrixCowboy[3][1] = terrain.getHeightTerrain(modelMatrixCowboy[3][0], modelMatrixCowboy[3][2]);
935         glm::mat4 modelMatrixCowboyBody = glm::mat4(modelMatrixCowboy);
936         modelMatrixCowboyBody = glm::scale(modelMatrixCowboyBody, glm::vec3(0.005, 0.005, 0.005));
937         modelAnimateCowboy.setAnimationIndex(0);
938         modelAnimateCowboy.render(modelMatrixCowboyBody);
939
940         modelMatrixRojo[3][1] = terrain.getHeightTerrain(modelMatrixRojo[3][0], modelMatrixRojo[3][2]);
941         glm::mat4 modelMatrixRojoBody = glm::mat4(modelMatrixRojo);
942         modelMatrixRojoBody = glm::translate(modelMatrixRojoBody, glm::vec3(0.0, 3.0, 0.0));
943         modelMatrixRojoBody = glm::scale(modelMatrixRojoBody, glm::vec3(0.002, 0.002, 0.002));
944         rojoModelAnimate.setAnimationIndex(numAnimacion);
945         rojoModelAnimate.render(modelMatrixRojoBody);
946
947         /*******************************************
```

Ahora bien, la manera en la que se implementó el cambio entre una animación y otra, fue detectando la tecla P y a partir de presionar aumentar una variable entera, de tal forma que podemos agregar más animaciones y visualizarlas sin ningún inconveniente, y en caso de llegar al límite de animaciones se reinicia.

```
605 // Cambiar animacion
606 if (glfwGetKey(window, GLFW_KEY_P) == GLFW_PRESS) {
607     numAnimacion++;
608     if (numAnimacion >= 2) {
609         numAnimacion = 0;
610     }
611 }
```

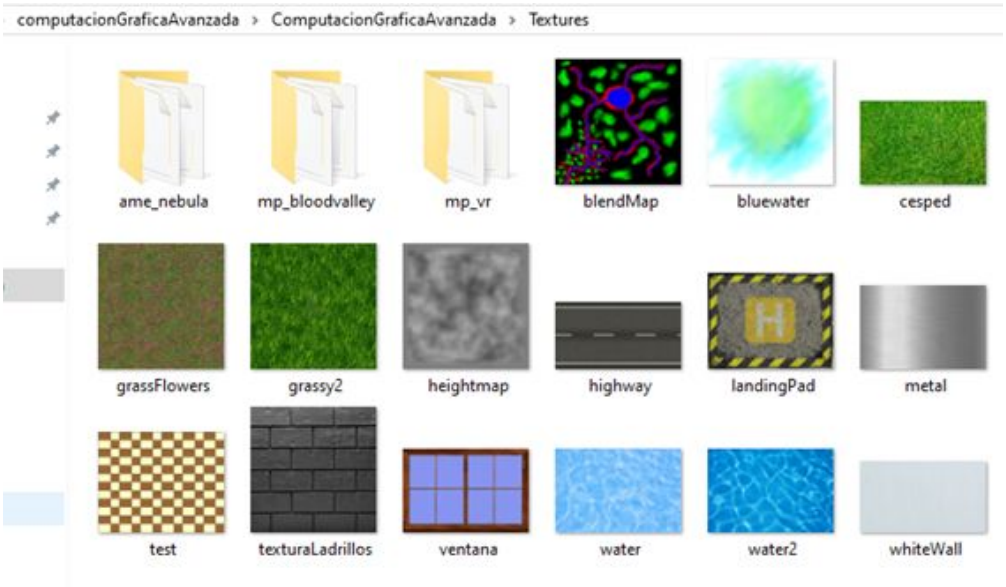
A continuación observamos la animaciones del modelo implementado, la captura de la izquierda nos muestra la animación 0, en donde se levantan y bajan los brazos, así como el movimiento de la cabeza, mientras que en la captura de la derecha observamos la animación 1, con el movimiento de giro del torso y de los pies.



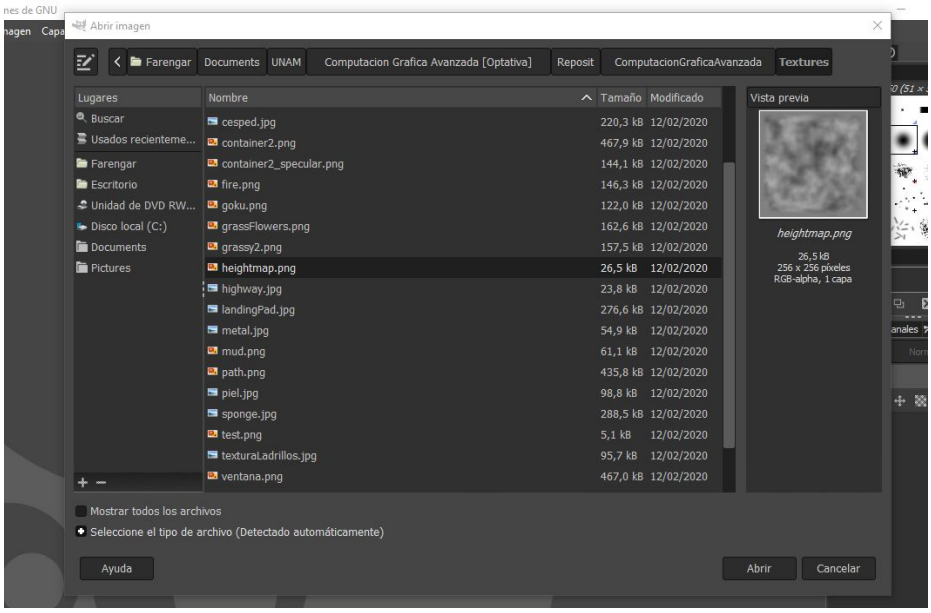
Terreno

El terreno corresponde a un mapa de alturas, las cuales son generadas a través de las variaciones entre el color blanco y negro que se aplican a una imagen.

Para la práctica presente, utilizaremos una imagen ya existente en el directorio *Textures* nombrada *heightImage*.



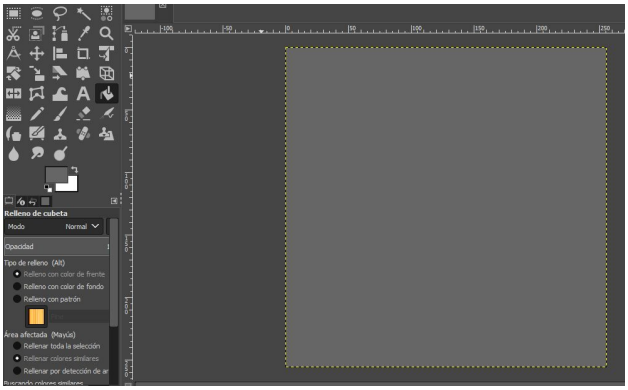
Dicha imagen, se editó en GIMP, al abrirla pudimos verificar sus propiedades entre las cuales la resolución que corresponde a 256*256 píxeles, como regla es necesario recordar que las imágenes siempre deben ser de un tamaño en potencia de 2 y de forma cuadrada, una vez que se tiene lista para editar la dejamos en un fondo blanco.

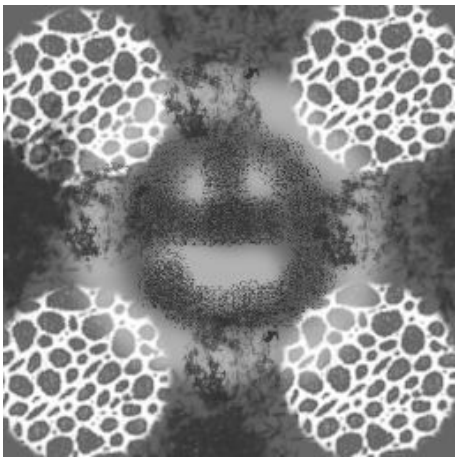


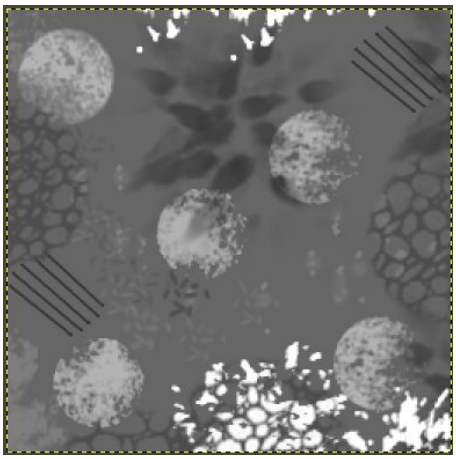
Par generar las diferentes alturas trabajaremos con la escala RGB, es necesario siempre seleccionarlos en la paleta de GIMP, dichas componentes tendrán el mismo valor para poder generar los tonos idóneos.

Para empezar colocaremos un tono más oscuro, para lo que definiremos que el valor de las componentes sea de 40.

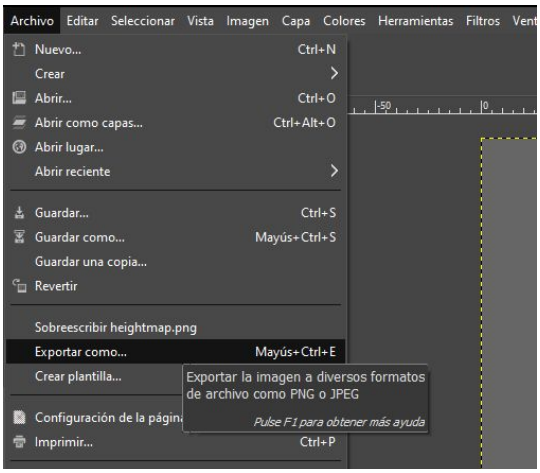
Una vez aplicado procederemos a ir dibujando lo que nosotros decidamos al indicar diferentes valores, recordando que mientras más cercano al blanco sea el tono, valores menores, se tendrán profundidades o cráteres, mientras que en caso de ser más cercanos al negro, valores mayores, se tendrán relieves elevados.







Una vez terminada de editar la imagen se exportará como imagen PNG.



Para poder hacer uso del mapa de alturas que representa un terreno, es necesario instanciar en el *main* del proyecto un objeto de tipo *terrain*, de tal forma que es necesario especificar ciertos parámetros, los cuales se explican a continuación:

Indicar el lugar donde al cual corresponderá el mapa en la escena, en este caso estos parámetros corresponden a (1,-1), lo que indica que está ubicado en el primer cuadrante, de tal forma que podemos manejar otro mapa de alturas solo bastará con especificar otro cuadrante.

Indicar el número de subdivisiones del mapa, el cual en este caso es 200*200.

Indicar la referencia en unidades de la altura máxima que se va a asignar al valor más oscuro y de ahí en proporción con los demás tonos, para poder generar las demá alturas en el mapa.

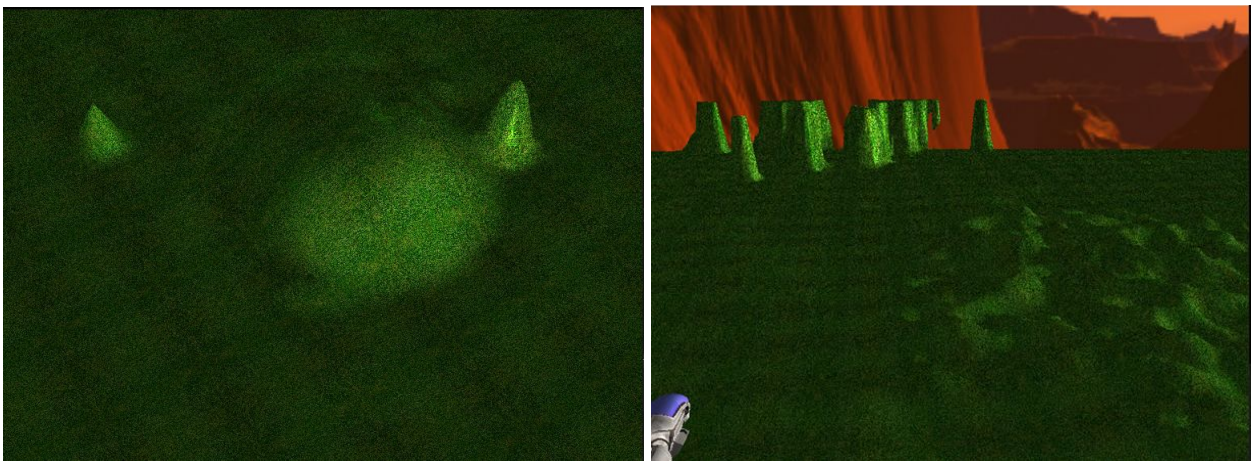
Indicar la ruta de la imagen del mapa de altura.

```
83    // Terrain model instance
84    Terrain terrain(-1, -1, 200, 8, "../Textures/heightmap.png");
85
```

Es importante mencionar, que al trabajar con terrenos, además de especificar un mapa de alturas, también estamos trabajando con texturas, de tal forma que también se especifica que textura se utilizara, las dimensiones, la escala de las UV y su posición.

```
824        // Se idica que textura emplear
825        glActiveTexture(GL_TEXTURE0);
826        // Se indica que la textura es de dos dimensiones y que se empleará siempre la misma
827        glBindTexture(GL_TEXTURE_2D, textureCespedID);
828        // Se indica que la textura se repetira para poder visulizarla mejor
829        shaderMulLighting.setVectorFloat2("scaleUV", glm::value_ptr(glm::vec2(150, 150)));
830        // Se posiciona la textura
831        terrain.setPosition(glm::vec3(75, 0, 75));
832        terrain.render();
833        shaderMulLighting.setVectorFloat2("scaleUV", glm::value_ptr(glm::vec2(0, 0)));
834        glBindTexture(GL_TEXTURE_2D, 0);
```

En la imagen que se muestra a continuación podemos observar la manera en que actúa un mapa de alturas y una textura, para crear la visualización de un terreno.



Cabe destacar, que al variar los valores en *shaderMulLighting*, podemos modificar su vista y la posición en la que se encuentra, de tal forma que podemos determinar el número de subdivisiones, la altura y posición en la que se centrara.

Es importante mencionar, que ahora es necesario indicar la presencia del terreno a los modelos que están en la escena, de tal forma que puedan desplazarse considerando el relieve del mismo, por lo que se especifica con el método siguiente:

```
940     modelMatrixCowboy[3][1] = terrain.getHeightTerrain(modelMatrixCowboy[3][0], modelMatrixCowboy[3][2]);
941     glm::mat4 modelMatrixCowboyBody = glm::mat4(modelMatrixCowboy);
```

Los valores *[3][0], [3][1]* y *[3][2]* corresponden a las respectivas coordenadas en la escena, lo que permitirá que el modelo se mueva de acuerdo a los relieves dados por las diferentes alturas del modelo.

De tal forma, que podemos visualizar nuestros modelos de la siguiente manera:

Conclusiones

Con las actividades realizadas en esta práctica, pudimos comprender y aplicar los pasos necesarios en la herramienta de Blender para crear un modelo 3D con diversas animaciones, proporcionandonos así un mecanismo para diseñar modelos más dinámicos y versátiles que puedan seR empleados en alguna escena.

Por otro lado, aprendimos e implementamos un mapa de alturas, de tal forma que comprendimos cada una de la peculiaridades para generar un mapa idóneo que pueda ser empleado, en este caso, en proyectos con OpenGL, lo que nos permitió diseñar un terreno más realista al complementar con una textura.

De igual manera, nos dimos cuenta, que es adecuado crear las diversas instancias y definir cada uno de sus especificaciones dentro del main de la respectiva práctica para poder agregar un nuevo modelo a la escena, y en este caso sin olvidar indicar la referencia del terreno para que este se adecue a él y nosotros podamos lo percibamos de forma adecuada.

Finalmente, podemos decir que el manejo de modelos más dinámicos y aspectos como relieves en el terreno, nos permitirán emplearlos en futuras prácticas, así como algún proyecto, de tal forma creamos y diseñamos escenas visualmente e interactivamente más llamativas, considerando que estamos a un bajo nivel de diseño en cuanto ambientes 3D por computadora.

Repositorio

- https://github.com/Sasfer/CGA2020-2