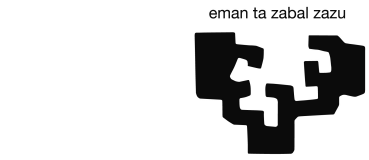



Universidad del País Vasco
Euskal Herriko Unibertsitatea

Grado en Ingeniería Informática

Computación

Proyecto de Fin de Grado

# Generación procedural de variaciones en modelos 3D

Autor

*Igor Santesteban Garay*


2017

# Resumen

El problema de partida de este proyecto es el de trasladar al mundo virtual la variabilidad que presentan los objetos del mundo real. Esta variabilidad juega un papel fundamental en la creación de escenas realistas, especialmente en aquellas que incluyen múltiples instancias de un mismo objeto (un bosque, un rebaño, una muchedumbre...). Sin embargo, tratar de capturar esta diversidad de forma manual es un proceso costoso tanto en tiempo de desarrollo como en espacio de memoria.

Este proyecto busca solventar estos dos inconvenientes mediante el uso de técnicas procedurales. Concretamente, se ha desarrollado un sistema capaz de generar variaciones de un modelo 3D de forma automática. Como generalizar el sistema a todo tipo de modelos excede las posibilidades de este proyecto, se ha decidido centrar el trabajo en la generación de variaciones de caballos. Estas variaciones afectan tanto a la textura como a la forma del modelo utilizado.

El sistema ha sido implementado en dos entornos diferentes. La primera implementación utiliza WebGL para llevar las funcionalidades del sistema a aplicaciones de gráficos 3D en la web. La segunda está integrada en Unity, un motor gráfico multiplataforma orientado al desarrollo de videojuegos.

# Índice general

# Índice de figuras

# 1. CAPÍTULO

# Introducción

## 1.1. Problema y objetivos

El problema de partida de este proyecto es el de **trasladar al mundo virtual la variedad que presentan los objetos en el mundo real**.

Este problema tiene especial interés en la **creación de escenas que incluyan múltiples instancias de un mismo objeto** (un bosque, un rebaño, una muchedumbre...), ya que replicar un único modelo da lugar a resultados poco convincentes (multitud de clones). Una manera de solventar este problema es mediante la creación de modelos adicionales que presenten pequeñas variaciones respecto al modelo original. Sin embargo, tratar de generar estas variaciones manualmente tiene algunos inconvenientes:

1. El tiempo requerido para modelar cada variación
2. El espacio de memoria que ocupa cada variación

El objetivo de este proyecto es tratar de solventar estos inconvenientes mediante el uso de técnicas procedurales. Concretamente, se propone la **implementación de un sistema capaz de generar variaciones de un modelo 3D de manera automática**. Las condiciones a satisfacer por el sistema son las siguientes:

- Ser capaz de generar variaciones en la forma y en la textura de un modelo 3D.

- Generar dichas variaciones de forma automática y en un intervalo de tiempo que permita su uso en aplicaciones de gráficos de tiempo real.

La forma en la que se generan las variaciones depende en gran medida del modelo utilizado. Como generalizar el sistema a todo tipo de modelos excede las posibilidades de este proyecto, se ha decidido centrar el trabajo en la **generación de variaciones del modelo de un caballo**. Este modelo tiene ciertas particularidades que lo hacen de gran interés para el proyecto:

- **Alta variabilidad en el pelaje**: el pelaje de los caballos puede presentarse en múltiples colores y con manchas de distintas formas y tamaños. Esto los convierte en un buen ejemplo para poner a prueba la flexibilidad del sistema en la generación de variaciones de texturas (Figura 1.1).

**Figura 1.1:** Ejemplos del tipo de mancha presentes en el pelaje de caballos reales (Fuente: *Wikimedia Commons*)

- **Modelo orgánico**: al tratarse del modelo de un animal es necesario que las variaciones de la forma se hagan respetando su anatomía, lo cual supone una dificultad añadida. El libro *"Simon & Schuster's Guide to Horses and Ponies"* [Bongianni, 1988] contiene abundante información sobre las características físicas de caballos de distintas razas.

La figura 1.2 muestra los resultados obtenidos con la implementación realizada, que se describe en detalle en los próximos capítulos de esta memoria. El modelo de partida se puede ver en la figura 1.3.

**Figura 1.2:** Algunos ejemplos de variaciones generadas por el sistema implementado

**Figura 1.3:** Modelo 3D del caballo utilizado en este proyecto

## 1.2. Antecedentes

### 1.2.1. Revisión previa a marcar objetivos

Durante la fase previa a la concreción del alcance se han revisado varias publicaciones que tratan problemas cercanos al tema escogido, pero que finalmente no han formado parte de este proyecto.

Una de estas publicaciones es *"Inverse Procedural Modeling of Trees"* [Stava et al., 2014], en la que se aborda el problema del modelado procedural de árboles. Concretamente, se propone un método que partiendo del modelo poligonal de un árbol es capaz de estimar los parámetros de un modelo procedural que genera resultados similares (Figura 1.4).

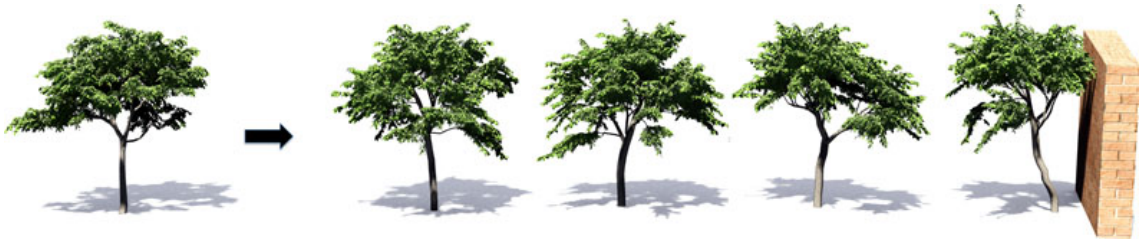


**Figura 1.4:** Imagen obtenida de *"Inverse Procedural Modeling of Trees"* que muestra el modelo poligonal de un árbol (izquierda) y cuatro variaciones generadas mediante un modelo procedural.

En *"Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination"* [Jain et al., 2012] se presenta un sistema capaz de generar nuevas formas a partir de otras ya existentes (Figura 1.5). La limitación es que las formas originales tienen que tener una estructura que posibilite su descomposición.

**Figura 1.5:** Imagen obtenida de *"Exploring Shape Variations by 3D-Model Decomposition and Part-based Recombination"* que muestra las variaciones que se pueden obtener combinando el modelo de una bicicleta con el de una moto.

Finalmente, en *"Procedural Facade Variations from a Single Layout"* [Bao et al., 2013] se aborda el problema de la generación de variaciones de fachadas de edificios. Este pro-

blema tiene especial interés si se enmarca dentro de la generación procedural de ciudades, ya que las variaciones de las fachadas contribuyen a un mayor realismo de las ciudades generadas (Figura 1.6).

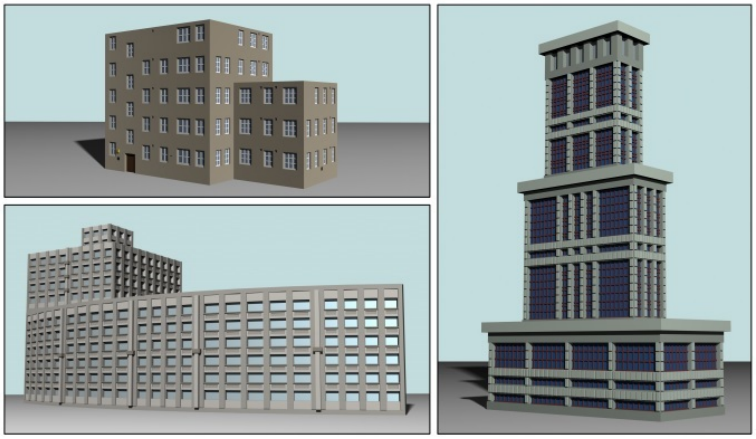


**Figura 1.6:** Imagen obtenida de *"Procedural Facade Variations from a Single Layout"* que muestra tres ejemplos de las fachadas que se pueden generar con el método propuesto.

### 1.2.2. Publicaciones en las que se apoya este trabajo

La mayor parte del trabajo realizado se apoya en publicaciones del campo de los gráficos por computador. Los principales temas que se han tratado durante el desarrollo de este proyecto han sido el procesado de imágenes, la generación procedural de texturas y técnicas de animación.

La principal referencia en el tema del procesado de imágenes ha sido *"Image Processing"* [Øyvind Kolås, 2005], que presenta algunas de las operaciones que se han utilizado en el filtrado de texturas (capítulo 2) y en el tratamiento de texturas procedurales (capítulo 3).

La mayor fuente de información sobre texturas procedurales ha sido el libro *"Texturing and Modeling: A Procedural Approach"* [Ebert et al., 2002], en particular los siguientes capítulos del mismo:

- Chapter 2: Building procedural textures
- Chapter 6: Practical methods for texture design
- Chapter 13: Real-time procedural solid texturing

Otras publicaciones consultadas respecto a este tema son *"Simplex noise demystified"* [Gustavson, 2005] y *"Understanding perlin noise"* [Biagioli, 2014], que explican el funcionamiento de dos algoritmos de generación de ruido (ruido Simplex y ruido de Perlin respectivamente). La forma en la que se han utilizado estos algoritmos se detalla en el capítulo 3 de esta memoria.

El capítulo 5 trata el problema de la generación de variaciones en la forma de un modelo 3D, para lo cual se propone la aplicación de técnicas similares a las utilizadas en animación. Las principales fuentes para entender el funcionamiento de estas técnicas han sido el manual de referencia de Blender [Blender Documentation Team, 2016] y el libro *"GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics"* [Randima, 2004], que trata este tema en su cuarto capítulo.

## 1.3. Recursos utilizados

### 1.3.1. Modelo 3D

El trabajo realizado en este proyecto se centra en el modelo de un caballo [Contreras, 2016] (Figura 1.3). La autoría de dicho modelo corresponde a las siguientes personas:

- Modelado: Tomasz Jurczyk
- Texturas: Didac Ruiz
- Esqueleto: Carlos Contreras

El modelo está distribuido bajo una licencia **Creative Commons BY 4.0** que permite su uso y adaptación siempre que se dé crédito de forma adecuada a los autores originales.

Como el fichero original utilizaba un formato propietario de Autodesk Maya, para poder trabajar con él ha sido necesario convertirlo al formato de Blender. Esto ha supuesto tener que hacer un nuevo esqueleto usando como referencia el de Contreras.

### 1.3.2. Librerías y aplicaciones

Durante el desarrollo del proyecto se han realizado **dos implementaciones distintas del sistema**: una que utiliza tecnologías web y otra integrada en un motor gráfico. Cada una de ellas ha requerido utilizar distintas herramientas.

### Apartado gráfico

La versión web del sistema utiliza la librería **Three.js** para todas las tareas relacionadas con el apartado gráfico (renderizado, iluminación, materiales, animaciones...). Three.js es una librería de Javascript de código abierto que utiliza WebGL para crear gráficos 3D en navegadores web.

La segunda versión del sistema está integrada en **Unity**, un motor gráfico multiplataforma orientado al desarrollo de videojuegos. Los aspectos generales de la aplicación (iluminación, materiales, animaciones...) se controlan desde el editor de Unity, mientras que las tareas más específicas se realizan mediante scripts.

### Herramientas de modelado 3D

Para las tareas que han requerido manipular el modelo 3D se ha utilizado **Blender**. Blender es un programa orientado a la creación de gráficos 3D que cuenta con herramientas de *"modelado, rigging, animación, simulación, renderizado, composición y captura de movimiento, e incluso edición de vídeo y creación de videojuegos"* [1].

Durante la fase inicial del proyecto también se ha utilizado **Maya**, ya que el fichero original del modelo 3D utilizaba un formato propietario de esta herramienta. Para poder instalar Maya ha sido necesario solicitar una licencia de estudiante.

### Entorno de desarrollo

El entorno de desarrollo utilizado para la versión web del sistema ha sido **WebStorm**, que está enfocado al desarrollo de aplicaciones en Javascript. Para la instalación de este programa ha sido necesario solicitar una licencia de estudiante.

Durante la fase de integración del sistema en Unity se ha utilizado **Visual Studio**, ya que el lenguaje de programación escogido para esta tarea ha sido C#.

### Navegadores

Las demos desarrolladas para mostrar las funcionalidades del sistema se han probado en varios navegadores. Los resultados obtenidos han sido los siguientes:

[1] Lista de funcionalidades obtenida de la propia web de Blender

1. Los navegadores **Opera**, **Mozilla Firefox** y **Google Chrome** ejecutan todas las demos correctamente.
2. El navegador **Microsoft Edge** ejecuta todas las demos pero las texturas procedurales se generan incorrectamente.
3. El navegador **Internet Explorer** no ejecuta ninguna de las demos.

Por ello se recomienda utilizar cualquiera de los tres primeros para acceder a las demos que se presentan en esta memoria.

#### Control de versiones

El sistema de control de versiones empleado ha sido **Git**. El repositorio que contiene las versiones del trabajo realizado está almacenado en **GitLab**, un gestor de repositorios gratuito y de código abierto.

Además de repositorios GitLab también permite alojar páginas web estáticas, lo cual se ha utilizado para hacer que las demos sean accesibles a través de Internet.

#### Redacción de la memoria

Esta memoria ha sido redactada con LaTeX, utilizando para ello la plantilla disponible en el sitio web del Grado en Ingeniería Informática de la UPV-EHU.

Para editar el documento se ha optado por utilizar **ShareLatex**, un editor online y gratuito de documentos LaTeX.

## 1.4. Diseño de la memoria

Esta memoria sigue una estructura similar a la seguida en el desarrollo del proyecto. El desarrollo ha consistido en varias fases en las que se han ido tratando los distintos aspectos del sistema. Los capítulos de esta memoria recogen el trabajo realizado en cada una de las etapas del desarrollo:

- Los capítulos 2 y 3 tratan el problema de la generación de variaciones de la textura de un modelo.

- El capítulo 4 presenta la estructura y funcionamiento del sistema generador de variaciones.
- El capítulo 5 aborda el problema de variar la forma de un modelo 3D.
- El capítulo 6 describe la manera en la que se ha automatizado la generación de las variaciones.
- El capítulo 7 recoge el trabajo realizado para integrar el sistema en aplicaciones creadas en Unity.
- Los capítulos 8 y 9 cierran la memoria mostrando los resultados obtenidos y las contribuciones realizadas por este proyecto.

Durante el desarrollo de cada uno de estos capítulos se ha evitado entrar en detalles de implementación en pos de ofrecer una visión más general del sistema. Esta decisión ha estado motivada por el interés en que el contenido que se presenta sea generalizable a cualquiera de las implementaciones realizadas.

Dicho esto, el código fuente de cada implementación puede consultarse públicamente desde los siguientes repositorios:

- **Implementación web del sistema**:
  https://gitlab.com/isantesteban/Procedural-Variations/
- **Implementación en Unity del sistema**:
  https://gitlab.com/isantesteban/Procedural-Variations-Unity/

Para complementar el contenido que se presenta en esta memoria se han desarrollado un total de **cuatro demos**. Las tres primeras se centran en aspectos concretos del trabajo realizado: filtros de color (Demo 1), texturas procedurales (Demo 2) y manipulación de la forma del modelo (Demo 3). La cuarta demo muestra los resultados que pueden obtenerse aplicando todas estas técnicas en conjunto.

El funcionamiento de estas demos se describe con mayor detalle en sus correspondientes capítulos, en los que se incluye también el enlace para acceder a ellas a través de Internet.

# 2. CAPÍTULO

## Filtros de color

El primer enfoque propuesto para generar variaciones de la textura de un modelo consiste en la aplicación de filtros. Estos filtros transforman la textura original modificando propiedades como el brillo, el contraste y la saturación (Figura 2.1).

El resultado del trabajo realizado en esta fase se presenta en la **Demo 1**, que permite generar de manera sencilla caballos con pelaje de distinto color (ver sección 2.4).

**Figura 2.1:** Algunas de las variaciones generadas automáticamente utilizando filtros

## 2.1. Implementación

Para implementar los filtros que se presentan en este capítulo se han utilizado *shaders*. Un *shader* es un programa diseñado para ejecutarse en una unidad de procesamiento gráfico o GPU, lo cual tiene varias ventajas respecto a la aplicación del filtro en la CPU:

- Las operaciones se realizan aprovechando el paralelismo de la GPU, lo que conlleva un menor tiempo de ejecución.
- La textura resultante se almacena directamente en la memoria de la GPU, de modo que no es necesario transferirla desde memoria principal para poder utilizarla.

El *shader* que implementa el filtro se ejecuta a nivel de píxel y afecta a toda la textura por igual. En la siguiente sección se describe cuáles son los parámetros del filtro y el efecto que tienen sobre la textura original.

## 2.2. Parámetros

### 2.2.1. Brillo

El brillo (*brightness*) afecta a la claridad general de la textura (Figura 2.2). Para modificar el brillo de la textura se suma una cantidad constante al color de los píxeles que la conforman:

$$color' = color + brillo$$

**Figura 2.2:** Resultado obtenido al modificar los niveles de brillo de la textura (-0.5, 0.0 y 0.5)

### 2.2.2. Contraste

Manipular el contraste (*contrast*) de una textura nos permite resaltar la diferencia entre las zonas oscuras y claras (Figura 2.3). Esta idea está recogida en la siguiente fórmula, que aumenta la diferencia del color de cada píxel respecto al color neutro:

$$color' = (color - 0.5) \times contraste + 0.5$$

**Figura 2.3:** Resultado obtenido al modificar el contraste de la textura (-0.5, 0.0 y 0.5)

### 2.2.3. Saturación

La saturación (*saturation*) está relacionada con la intensidad del color de los píxeles que forman la textura (Figura 2.4).

**Figura 2.4:** Resultado obtenido al modificar la saturación de la textura (-1 y 0)

Para modificar la saturación de la textura se ha realizado una conversión del espacio de color RGB al espacio HSV, en el que cada componente representa una propiedad distinta: tonalidad (*Hue*), saturación (*Saturation*) y valor (*Value*). Trabajar en el espacio HSV

permite expresar los cambios en la saturación como una simple suma a uno de los componentes del color.

Para realizar la conversión de RGB a HSV (y viceversa) se han utilizado las funciones implementadas por Sam Hocevar en *"Fast branchless RGB to HSV conversion in GLSL"*.

### 2.2.4. Composición de parámetros

Como modificar el brillo y el contraste por separado puede dar lugar a resultados poco naturales, se ha introducido una propiedad adicional (*lightness*) definida a partir de estas dos. Esta propiedad controla la claridad de la textura modificando conjuntamente el brillo y el contraste (la relación entre los parámetros se ha establecido a partir de la observación):

$$brightness = 0.25 \times lightness$$

$$contrast = 0.40 \times lightness + 1.0$$

De este modo se consiguen dos cosas:

- Reducir el número de parámetros de los que depende el sistema.
- Eliminar del conjunto de combinaciones posibles aquellas que dan resultados poco naturales.

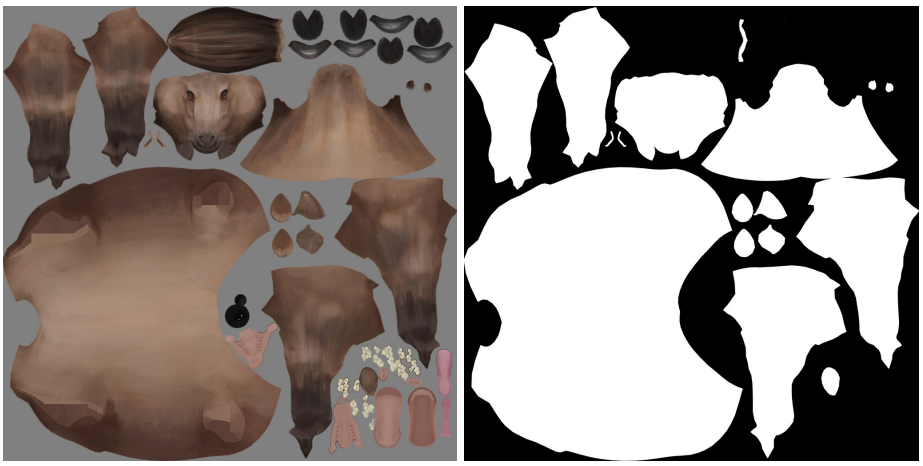
## 2.3. Uso de máscaras

Los filtros presentados en este capítulo tienen la limitación de que afectan a toda la textura por igual. Este inconveniente se ha solventado dando la opción de utilizar máscaras que delimiten las zonas de la textura sobre las que se aplica el filtro.

Una máscara es una textura en blanco y negro utilizada para indicar en qué partes de la textura se puede aplicar el filtro (blanco) y en cuáles no (negro). Por ejemplo, utilizar una máscara nos puede servir para evitar que el filtro se aplique sobre ciertas partes del caballo, como la cola, las pezuñas o los ojos (Figura 2.5).

Aunque estas máscaras pueden ser generadas manualmente en una herramienta de edición de imágenes, el verdadero interés de este proyecto son las máscaras generadas procedural-mente. Justamente este es el problema que se aborda en el próximo capítulo, que estudia

el uso de texturas procedurales para recrear las manchas presentes en el pelaje de algunos caballos.



**(a)** Textura original **(b)** Máscara

**Figura 2.5:** Ejemplo de una máscara generada manualmente que identifica las partes correspondientes al pelaje del caballo

## 2.4. Demo

**Enlace a la demo**: http://isantesteban.com/tfg/demo1/

La demo desarrollada en esta fase del proyecto permite aplicar un filtro sobre la textura del caballo. Se trata de una demo interactiva en la que el usuario puede manipular los parámetros del filtro (*Lightness* y *Saturation*) para generar caballos de distinto color.

El valor de estos parámetros se modifica desde el panel de control de la demo, el cual incluye opciones adicionales para rotar el modelo (*Rotate model*), aleatorizar el color del caballo (*Randomize*) y restaurar el valor de los parámetros (*Clear*).

# 3. CAPÍTULO

## Texturas procedurales

Este capítulo estudia la creación y aplicación de texturas procedurales. El objetivo de estas texturas es imitar los distintos tipos de manchas presentes en el pelaje de algunos caballos (Figura 3.1) y ampliar así el rango de variaciones que el sistema es capaz de generar.

El resultado del trabajo realizado en esta fase está disponible en la **Demo 2**, que permite generar automáticamente distintos tipos de texturas a partir de unos parámetros que el usuario puede modificar (ver sección 3.4).

**Figura 3.1:** Ejemplos del tipo de mancha presentes en el pelaje de algunos caballos

## 3.1. Implementación

Al igual que los filtros del capítulo anterior, las texturas procedurales han sido implementadas mediante *shaders*. El *shader* desarrollado utiliza un algoritmo de generación de ruido para producir texturas que se asemejan a manchas (Figura 3.2).

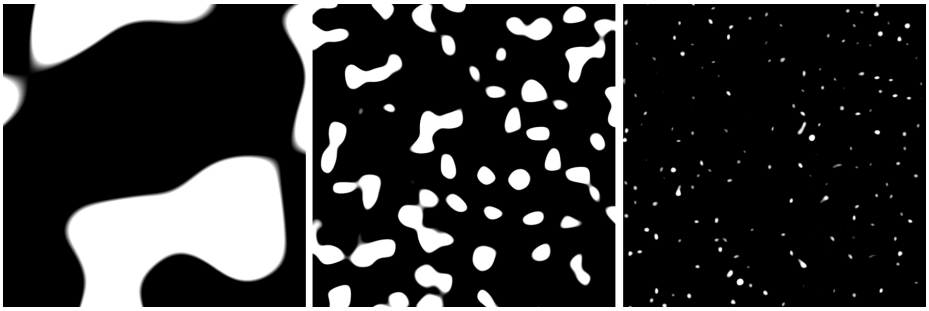


**Figura 3.2:** Algunos ejemplos de las texturas generadas proceduralmente

### 3.1.1. Generación de ruido

El método de generación de ruido escogido es el del **ruido simplex** (Figura 3.3). Este método fue desarrollado por Ken Perlin para solventar algunas de las limitaciones del ruido de Perlin, otro algoritmo de generación de ruido creado por él mismo años atrás.

**Figura 3.3:** Sección 2D obtenida a partir de ruido simplex 3D

Una de las características del ruido simplex es que se puede generalizar a cualquier dimensión. El *shader* implementado utiliza ruido simplex 3D, ya que de este modo se evitan

las distorsiones y discontinuidades que pueden surgir al aplicar una textura 2D sobre la superficie de un modelo 3D.

La autoría de la función utilizada para generar ruido le corresponde a Ashima Arts, que permite su uso bajo las condiciones de una licencia MIT. Esta función recibe de entrada la posición de un vértice en el espacio del modelo y devuelve un valor en el rango $[-1,1]$:

$$noise = snoise(position)$$

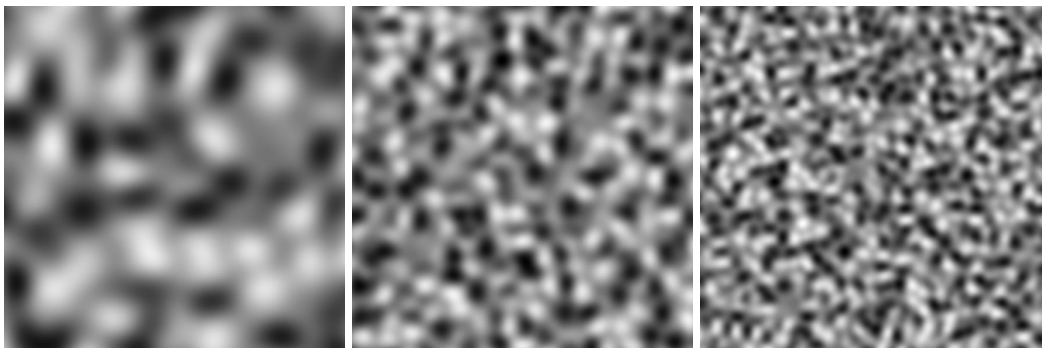
## 3.2. Parámetros

A continuación se presentan los parámetros utilizados para controlar el aspecto del ruido generado. Estos parámetros se pueden modificar conjuntamente para obtener texturas que se asemejen a manchas de distintas formas y tamaños.

### 3.2.1. Frecuencia

La frecuencia (*frequency*) es un factor que multiplica al valor de entrada de la función de generación de ruido. La frecuencia controla la escala del ruido generado (Figura 3.4):

$$noise = snoise(frequency \times position)$$



**(a)** Frecuencia = 1.0 **(b)** Frecuencia = 1.5 **(c)** Frecuencia = 2.0

**Figura 3.4:** Texturas obtenidas utilizando distintas frecuencias

### 3.2.2. Desplazamiento

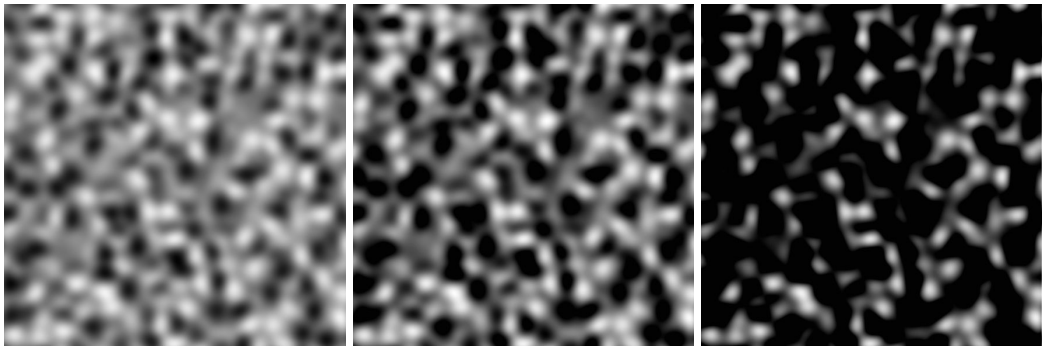
El desplazamiento (*displacement*) es una cantidad constante sumada al valor de entrada de la función de generación de ruido. Modificar el desplazamiento nos permite aleatorizar el ruido generado sin alterar sus características visuales:

$$noise = snoise(frequency \times position + displacement)$$

### 3.2.3. Umbral

La aplicación de un umbral (*threshold*) permite anular las partes de la textura cuyo valor esté por debajo de un valor escogido (Figura 3.5). Esta operación se define de la siguiente manera:

$$noise' = \begin{cases} noise & noise \geq threshold \\ 0 & noise < threshold \end{cases}$$



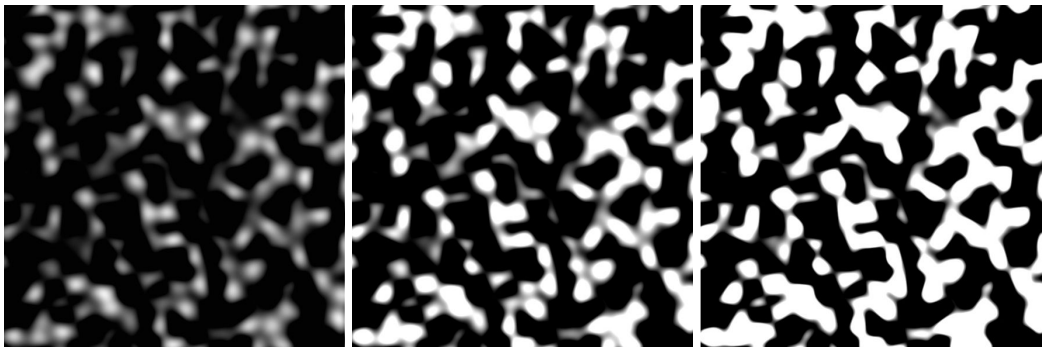
**(a)** Umbral = 0 **(b)** Umbral = 0.25 **(c)** Umbral = 0.5

**Figura 3.5:** Resultados obtenidos aplicando distintos umbrales

### 3.2.4. Nitidez

La nitidez (*sharpness*) de una textura está relacionada con el nivel de definición de las zonas no-nulas (Figura 3.6). Para modificar la nitidez se multiplica toda la textura por un factor constante, restringiendo el resultado de dicha multiplicación al rango $[0, 1]$:

$$noise' = clamp(sharpness \times noise,\ 0.0,\ 1.0)$$



**(a)** Nitidez = 1 **(b)** Nitidez = 5 **(c)** Nitidez = 10

**Figura 3.6:** Texturas obtenidas utilizando distintos niveles de nitidez

### 3.2.5. Distorsión

La distorsión (*distortion*) introduce ruido adicional con el fin de conseguir contornos más irregulares y con mayor apariencia de pelo (Figura 3.7).

**(a)** Distorsión = 0 **(b)** Distorsión = 1 **(c)** Distorsión = 2

**Figura 3.7:** Texturas obtenidas utilizando distintos niveles de distorsión

### 3.2.6. Alcance

Además de controlar la apariencia de las texturas procedurales también es posible controlar su alcance. Este control se realiza asignando valores a los siguientes parámetros: *max_x, max_y, max_z, min_x, min_y*, *min_z*.

En conjunto, estos parámetros definen un ortoedro que delimita el espacio que abarca la textura procedural. Para evitar un corte abrupto en los límites de dicho espacio se ha

optado por incrementar progresivamente el umbral que se aplica a la textura. De este modo se consigue que la textura desaparezca de forma gradual y que el resultado final sea más natural (Figura 3.8).

**(a)** Sin límites **(b)** Con límites

**Figura 3.8:** Comparación entre una textura procedural aplicada sobre todo el modelo y otra con alcance limitado (límites resaltados en color)

El alcance de las texturas procedurales también se puede controlar utilizando máscaras. Sin embargo, esta opción se ha habilitado únicamente para los casos en los que se necesita mayor precisión (por ejemplo, para evitar que las manchas se generen sobre las pezuñas o la cola del caballo), ya que el uso de máscaras conlleva un trabajo manual que es preferible evitar.

## 3.3. Integración con la textura original

Una vez generada la textura procedural el siguiente paso consiste en integrarla en la textura original del modelo. Para conseguirlo se han explorado dos alternativas distintas.

La primera de estas alternativas ha consistido en **superponer la textura procedural utilizando un color uniforme**. Los resultados obtenidos con este método son muy pobres puesto que la textura procedural carece del nivel de detalle de la textura original. Esto da lugar a que las manchas parezcan manchas de pintura en vez de las manchas que buscamos.

La segunda alternativa, que es la que se ha utilizado finalmente, consiste en **combinar las texturas procedurales con los filtros presentados en el capítulo anterior**. Utilizando este método las texturas procedurales actúan como máscaras, es decir, indican las

zonas sobre las que se debe aplicar el filtro. De este modo se consigue generar manchas manteniendo el nivel de detalle de la textura original.

En el próximo capítulo se describe el sistema generador de variaciones, que permite combinar múltiples filtros y texturas procedurales para obtener variaciones como las que se presentan en la figura 3.9.

**Figura 3.9:** Un ejemplo de las variaciones que se pueden generar de forma automática combinando múltiples texturas procedurales

## 3.4. Demo

**Enlace a la demo**: http://isantesteban.com/tfg/demo2/

La demo desarrollada en esta fase del proyecto permite aplicar una textura procedural sobre el modelo de un caballo. Se trata de una demo interactiva en la que el usuario puede modificar los siguientes parámetros de la textura:

- **Parámetros de control del aspecto**: *frequency, displacement, threshold, sharpness* y *distortion*.
- **Parámetros de control del alcance**: *max_x, max_y, max_z, min_x, min_y* y *min_z*.

En el caso de la frecuencia (*frequency*) también se da la opción de utilizar un valor distinto en cada uno de los ejes XYZ. Estos parámetros se modifican desde el panel de control de la demo, que incluye opciones adicionales para visualizar la textura en 2D (*View texture*) y rotar el modelo (*Rotate model*).

# 4. CAPÍTULO

# Sistema generador de variaciones

En este capítulo se introduce la estructura y el funcionamiento del **sistema generador de variaciones**. Esta introducción incluye también la manera en la que el sistema integra las técnicas presentadas en capítulos anteriores: los filtros y las texturas procedurales.

## 4.1. Estructura del sistema

El sistema para generar variaciones está compuesto por los siguientes elementos:

- El modelo del que se quieren generar variaciones
- Un conjunto de modificadores

Los **modificadores** (*modifiers*) definen las transformaciones que el sistema debe aplicar al modelo y pueden afectar tanto a su textura como a su forma. A su vez, estos modificadores se componen de un conjunto de **parámetros de control** que regulan el efecto del modificador sobre el modelo.

Este planteamiento permite dividir el comportamiento del sistema en componentes más fáciles de gestionar. El resultado es un sistema cuya estructura se asemeja a la de una **cadena de montaje**, en la que cada trabajador (cada modificador) se encarga de una parte específica del proceso de producción (la generación de variaciones). La figura 4.1 muestra un esquema de cómo está estructurado el sistema:

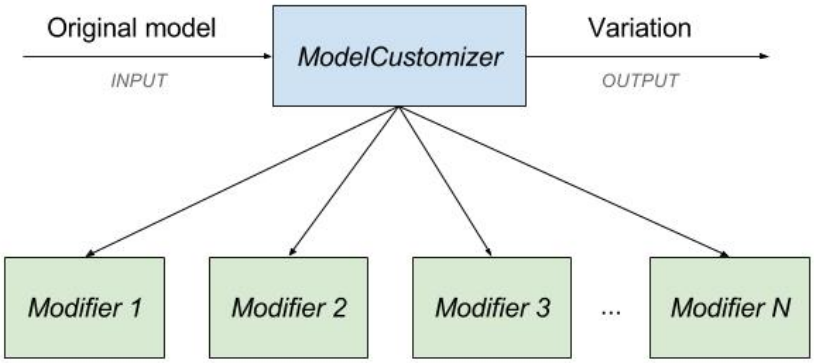



**Figura 4.1:** Estructura del sistema diseñado para generar variaciones de un modelo 3D

La implementación del sistema está formada por varias clases. La principal de estas clases es `ModelCustomizer`, que se encarga del procesado del modelo original y de la aplicación de los modificadores. Estos modificadores son instancias de la clase `Modifier` o de clases derivadas de ella, como `TextureModifier` y `SkeletonModifier`.

Los filtros de color presentados en el capítulo 2 se aplican utilizando modificadores de la clase `TextureModifier`, los cuales pueden contener un número ilimitado de texturas procedurales. Las texturas procedurales son creadas a partir de la clase `ProceduralTexture`.

En el capítulo 5 se presentan los modificadores de la clase `SkeletonModifier`, que son utilizados para generar variaciones de la forma de un modelo.

## 4.2. Funcionamiento del sistema

### 4.2.1. Aplicación de modificadores

Para generar una variación del modelo de entrada **el sistema aplica los modificadores secuencialmente**. Las operaciones que se realizan para aplicar cada uno de los modificadores dependen de la clase a la que pertenezcan. En el caso de los modificadores de texturas las operaciones realizadas son las siguientes:

1. Renderizar las texturas procedurales
2. Aplicar el filtro utilizando como máscara las texturas procedurales renderizadas

Para que las texturas procedurales puedan ser utilizadas como máscaras es necesario que su formato sea el de una textura 2D. En el siguiente apartado se describe cómo generar una textura 2D convencional a partir de una textura procedural.

### Renderizado de texturas procedurales

En el capítulo 3 de esta memoria se ha presentado un método para crear texturas a partir de ruido simplex 3D. La principal ventaja de generar las texturas en el espacio 3D es **evitar que se produzcan distorsiones** al aplicarlas sobre la superficie del modelo. Sin embargo, una vez generadas es posible utilizar el mapeado UV del modelo para obtener texturas 2D que mantengan el mismo aspecto.

El **mapeado UV** describe la manera en la que una textura 2D se proyecta sobre un modelo 3D. Para renderizar las texturas procedurales utilizando esta información se han hecho varios cambios en el *shader* que las genera:

- El resultado producido por el *shader* se almacena en un *buffer* en vez de dibujarse por pantalla. El tamaño de dicho *buffer* depende de la resolución escogida para las texturas, que por defecto se ha establecido en $1024 \times 1024$ píxeles.
- Para indicar la posición del *buffer* sobre la que dibujar cada píxel se utilizan las coordenadas UV asignadas a los vértices del modelo. En WebGL esto requiere convertir las coordenadas de textura a coordenadas de dispositivo normalizado (NDC) y asignar el valor resultante a la variable *gl_Position*. Para realizar esta conversión basta con transformar el rango de las coordenadas UV de [0, 1] a [-1, 1]:

$$gl\_Position = vec4(2.0 \times uv - 1.0,\ 0.0,\ 1.0);$$

Las texturas generadas de esta manera presentan un problema adicional, ya que al aplicarlas sobre el modelo surgen pequeñas discontinuidades (Figura 4.2). Para solucionarlo se ha optado por aplicar un filtro que expande los contornos de cada sección de la textura (Figura 4.3).

Este filtro realiza los siguientes pasos:

1. Seleccionar los píxeles correspondientes a los contornos de las distintas secciones del mapeado UV. Usando de ejemplo la figura 4.2, los píxeles a seleccionar serían los píxeles rojos adyacentes a uno o más píxeles blancos.

2. Asignar a los píxeles seleccionados el valor medio de sus adyacentes no-nulos. Los píxeles no-nulos son aquellos que están asignados a alguna sección del mapeado UV (en la figura 4.2 serían los píxeles que están resaltados de blanco).

**Figura 4.2:** Ejemplo de las discontinuidades que se forman al aplicar una textura renderizada (fondo coloreado de rojo para resaltar mejor el problema)

**Figura 4.3:** Resultado tras el filtrado de la textura renderizada

### 4.2.2. Gestión de recursos

Idealmente nos gustaría que el sistema pudiese aplicar un **número ilimitado de modificadores** y que estos puedan utilizar un **número ilimitado de texturas procedurales**. Aunque los recursos de los que dispone el sistema son limitados, una gestión adecuada de los mismos puede acercarnos a este objetivo.

#### Gestión de memoria

El principal recurso que consume el sistema es **memoria de la GPU**. Además de la textura asignada a cada variación el sistema también genera múltiples texturas auxiliares (máscaras y resultados de pasos intermedios). Estas texturas auxiliares no tienen ningún valor una vez creada la textura final y por tanto pueden ser liberadas de memoria.

Para llevar un seguimiento de las texturas se ha implementado un **gestor de recursos** (`ResourceManager`), a través del cual el sistema crea todas las texturas que necesita. Por cada una de las texturas generadas el gestor monitoriza el tiempo que lleva en activo y si está asignada a algún material.

Las texturas que no están asignadas a ningún material son liberadas cuando su tiempo de vida supera el tiempo de vida límite (establecido en 500ms). Esta ventana de tiempo es suficiente para que la generación de variaciones no se vea afectada.

El gestor de recursos también dispone de una **pila de texturas libres**. La finalidad de esta pila es evitar que las texturas se tengan que crear de cero cada vez que se necesitan, ya que ello supone un perjuicio para el rendimiento de la aplicación. Cuando el gestor detecta que una textura ya no es necesaria, en vez de liberarla de memoria, la almacena en la pila hasta que el sistema la vuelva a necesitar. Esta pila tiene un tamaño máximo preestablecido, que en el caso de la demo final es de siete texturas.

#### Unidades de texturas

Las unidades de textura son componentes de la GPU. El número de unidades de texturas determina cuántas texturas pueden utilizarse durante la ejecución de un *shader*, por consiguiente, también **limita el número de máscaras que se pueden utilizar** en la aplicación de un filtro.

Esta limitación depende del hardware sobre el que se ejecute el programa, que en el caso de WebGL abarca desde móviles y tablets hasta ordenadores de sobremesa. El sitio web `www.webglstats.com` ofrece estadísticas de interés sobre los dispositivos que soportan WebGL[1]:

- El 100 % de los dispositivos disponen de al menos 8 unidades de textura
- El 84.8 % de los dispositivos disponen de al menos 16 unidades de textura
- El 4.6 % de los dispositivos disponen de 32 unidades de textura

Partiendo de que cada máscara ocupa una unidad de textura, si al aplicar un filtro superásemos las 16 unidades para el 84.8 % de los potenciales usuarios la aplicación dejaría de funcionar correctamente. La solución propuesta para solventar este problema consiste en **realizar un paso adicional que combine todas las máscaras en una**, de manera que el filtro se pueda aplicar utilizando una única unidad de textura.

Este paso se ha implementado en el método `blend()` de la clase `TextureBlender`. Si el número de texturas a combinar excede las capacidades de la GPU este método divide el proceso en varias iteraciones, de modo que en cada una se combinen tantas texturas como sea posible. El número total de iteraciones necesarias para combinar $n$ texturas utilizando $m$ unidades de textura será:

$$\text{Nº Iteraciones} = \left\lceil \frac{n-1}{m-1} \right\rceil$$

Para calcular el número de iteraciones hay que tener en cuenta que tras la primera iteración es necesario reservar una unidad de textura para almacenar el resultado de la iteración anterior. Por ejemplo, si una GPU dispone de 16 unidades de textura ($m = 16$):

$$\text{Si } n = 16 \text{ entonces } \left\lceil \frac{16-1}{16-1} \right\rceil = 1 \text{ iteración}$$

$$\text{Si } n = 32 \text{ entonces } \left\lceil \frac{32-1}{16-1} \right\rceil = \lceil 2.01\overline{6} \rceil = 3 \text{ iteraciones}$$

Para guardar el resultado de cada iteración se utilizan 2 buffers que se van alternando, ya que no es posible leer y escribir en un mismo buffer simultáneamente.

[1] Datos consultados el 19 de febrero de 2017

Implementación del método `blend()` en Javascript:

```
blend(textures) {

    if (textures.length === 0) return null;
    if (textures.length === 1) return textures[0];

    // Hacer una copia del array original
    let texturesToBlend = textures.slice();

    // Obtener el número de unidades de texturas disponibles
    let numTexUnits = GraphicUtilities.getCapabilities().maxTextures;

    // Crear dos buffers para guardar el resultado de cada iteración
    let firstRenderTarget = ResourceManager.getRenderTarget();
    let secondRenderTarget = ResourceManager.getRenderTarget();
    let currentRenderTarget = firstRenderTarget;

    while (texturesToBlend.length > 1) {

        // Seleccionar las texturas a combinar
        let iterationTextures = texturesToBlend.splice(0, numTexUnits);

        // Actualizar el material con las texturas seleccionadas
        this.updateMaterial(iterationTextures);

        // Combinar las texturas
        GraphicUtilities.renderQuad(this.textureBlendingMaterial, currentRenderTarget);

        // Añadir el resultado obtenido al comienzo del array de texturas
        texturesToBlend.unshift(currentRenderTarget.texture);

        // Intercambiar los buffers para la siguiente iteración
        if (currentRenderTarget.uuid === firstRenderTarget.uuid) {
            currentRenderTarget = secondRenderTarget;
        } else {
            currentRenderTarget = firstRenderTarget;
        }
    }

    // Devolver el resultado obtenido
    return texturesToBlend[0];
}
```

# 5. CAPÍTULO

# Deformación de mallas

En este capítulo se estudian modificadores que permitan **generar variaciones de la malla poligonal del modelo**. En capítulos anteriores se han estudiado métodos para variar la textura (piel) de un modelo, en este buscamos variar su forma.

El objetivo es conseguir variaciones más complejas de las que se podrían obtener realizando un simple escalado. Para ello, el requisito de partida será **disponer de un esqueleto** que permita deformar la malla del modelo.

La **Demo 3** muestra los resultados que pueden obtenerse aplicando el método propuesto sobre el modelo de un caballo. La forma del modelo se puede manipular en función de tres parámetros: la altura del caballo, su peso, y la longitud de su cola (ver sección 5.2).

## 5.1. Esqueletos

En el campo de los gráficos 3D, un esqueleto es una estructura formada por un conjunto de huesos organizados jerárquicamente. Los esqueletos permiten manipular la malla de un modelo de forma similar a cómo un esqueleto real controla el movimiento de un cuerpo:

- Cada hueso tiene influencia sobre un conjunto de vértices, de modo que las transformaciones aplicadas sobre el hueso se aplican también sobre dichos vértices.
- Las transformaciones aplicadas sobre un hueso se propagan a los huesos inferiores de la jerarquía.

Esta técnica es frecuentemente utilizada en el campo de la animación, ya que proporciona una manera intuitiva de controlar el movimiento de un modelo a lo largo de una secuencia de fotogramas.

En nuestro caso utilizar un esqueleto nos va a servir para poder modificar la forma del modelo de forma coherente con su anatomía.

### 5.1.1. Creación de un esqueleto

La creación de un esqueleto consta de dos pasos: la creación de la estructura (*rigging*) y la asignación del esqueleto a la malla (*skinning*). Ambas tareas han sido realizadas en Blender con ayuda del manual de referencia [Blender Documentation Team, 2016].

El esqueleto utilizado en este proyecto está compuesto por 48 huesos que controlan las distintas partes del cuerpo de un caballo: patas, dorso, cabeza, cuello, cola, orejas... (Figura 5.1).

Para que las transformaciones aplicadas sobre el esqueleto afecten a la malla es necesario tener una manera de relacionarlos. Esta relación se define de la siguiente manera [Randima, 2004]:

- Cada vértice de la malla puede estar influenciado por uno o más huesos del esqueleto.
- Cada uno de los huesos que influencian a un determinado vértice tiene asignado un peso. Este peso determina el grado de influencia del hueso sobre dicho vértice.
- La posición de un vértice se calcula a partir de la suma ponderada de las transformaciones aplicadas a los huesos que lo influencian.

La asignación de pesos es una tarea tediosa y que requiere especial cuidado en las zonas en las que se producen contracciones de los músculos. Afortunadamente Blender dispone de herramientas para realizar esta tarea con mayor facilidad, como la asignación automática de pesos y el modo *Weight Paint*.

### 5.1.2. Generación de variaciones de un esqueleto

Una vez creado el esqueleto el siguiente paso consiste en generar variaciones del mismo. Los modificadores utilizados para ello son de la clase `SkeletonModifier`, que permiten

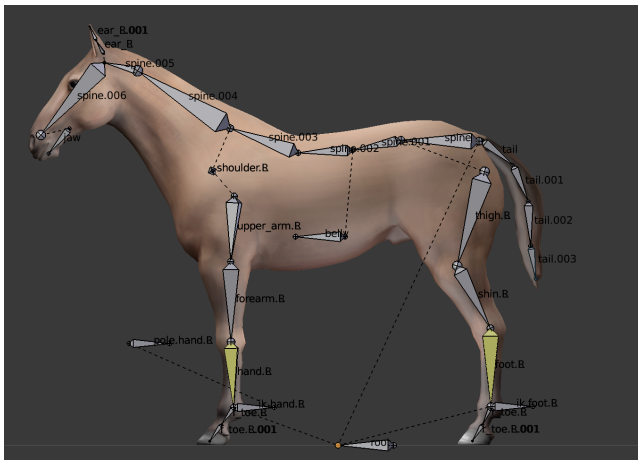



**Figura 5.1:** Esqueleto creado en Blender para manipular la forma de un caballo

combinar transformaciones sencillas (escalados, traslaciones y rotaciones) para obtener transformaciones más complejas. Los métodos utilizados para ello son los siguientes:

`scaleBone(`*`name, scale, scaleChildren`*`)`

- *name* → nombre del hueso sobre el que aplicar el escalado
- *scale* → vector que contiene el factor de escalado de cada eje
- *scaleChildren* → valor booleano para indicar si el escalado debe aplicarse a los huesos inferiores de la jerarquía

`translateBone(`*`name, position`*`)`

- *name* → nombre del hueso a trasladar
- *position* → vector que contiene la posición a la que trasladar el hueso

`rotateBone(`*`name, rotation`*`)`

- *name* → nombre del hueso a rotar
- *rotation* → vector que contiene la rotación a realizar en cada eje

Las transformaciones definidas mediante estos métodos se almacenan en un array de transformaciones. Cuando el sistema tiene que generar una variación del modelo estas transformaciones se aplican sobre el esqueleto en el mismo orden en el que fueron introducidas.

## 5.2. Demo

**Enlace a la demo**: http://isantesteban.com/tfg/demo3/

La demo desarrollada en esta fase del proyecto utiliza los modificadores presentados para manipular la forma de un caballo. Se trata de una demo interactiva en la que el usuario tiene control sobre el valor de tres parámetros: la altura del caballo, su peso, y la longitud de su cola (Figura 5.2).

Estos valores se modifican desde el panel de control de la demo, el cual incluye opciones adicionales para visualizar el esqueleto (*Show skeleton*), rotar el modelo (*Rotate model*), aleatorizar la forma del caballo (*Randomize*) y restaurar el valor de los parámetros (*Clear*).

Para implementar esta demo se han utilizado tres modificadores (uno por grado de libertad) que actúan sobre varios huesos del esqueleto del modelo. Concretamente, las partes del modelo afectadas por cada modificador son las siguientes:

- **Modificador altura**: realiza un escalado de las patas frontales y traseras
- **Modificador peso**: realiza un escalado del vientre, el cuello y las patas
- **Modificador cola**: realiza un escalado de la cola

**Figura 5.2:** Resultados obtenidos con distintos valores de altura, peso y longitud de cola:
Izquierda: altura -10 %, peso -20 %, longitud cola -20 %
Derecha: altura +10 %, peso +20 %, longitud cola +20 %

# 6. CAPÍTULO

## Automatización y aleatoriedad

Este capítulo describe la manera en la que el sistema controla los parámetros implicados en la generación de variaciones de un modelo (los expuestos en capítulos anteriores).

El objetivo es que el sistema sea capaz de generar variaciones de forma automática y que los modelos resultantes se asemejen a las variaciones existentes en el mundo real. Para conseguirlo el sistema permite:

- Aleatorizar parámetros en función de distintas distribuciones de probabilidad
- Establecer relaciones entre parámetros

La **demo final** (que se presenta en el capítulo 8) muestra el resultado del trabajo realizado en esta fase. Esta demo consiste en un desfile de caballos en el que continuamente se introducen nuevas variaciones. Las variaciones se generan de forma automática gracias a las herramientas presentadas en este capítulo.

### 6.1. Parámetros del sistema

Tal y como se recoge en capítulos anteriores, el sistema se compone de los siguientes elementos:

- El modelo del que se quieren generar variaciones
- Un conjunto de modificadores que definen cómo generar dichas variaciones

Cada modificador dispone de un conjunto de **parámetros de control** que determinan el efecto del modificador sobre el modelo. A continuación se presenta una lista completa de estos parámetros (agrupados en función de la clase a la que pertenecen):

- **Modifier**: *enabled*
- **TextureModifier**: *lightness*, *saturation*
- **SkeletonModifier**: *location*, *scale*, *rotation*
- **ProceduralTexture**: *frequency, displacement, threshold, sharpness*, *distortion*

Las texturas procedurales tienen además los siguientes parámetros para controlar su alcance: *max_x, max_y, max_z, min_x, min_y*, *min_z*.

## 6.2. Aleatorización de parámetros

La mayoría de los parámetros que maneja el sistema son numéricos, y para poder generar variaciones es necesario que algunos de ellos tomen valores pseudo-aleatorios. Además, dependiendo del tipo de parámetro puede convenir que estos valores se generen a partir de **distintas distribuciones de probabilidad**.

Las siguientes funciones permiten redefinir el comportamiento de un parámetro para que devuelva un número distinto cada vez que se accede a su valor. Estos números pueden generarse a partir de una distribución uniforme o una distribución normal:

`randomizeFloat(`*`source, property, min, max`*`)`

- *source* → objeto que contiene el parámetro a aleatorizar
- *property* → parámetro a aleatorizar utilizando una distribución uniforme
- *min* → valor mínimo del parámetro
- *max* → valor máximo del parámetro

`randomizeFloatNormal(`*source, property, mean, sd*`)`

- *source* → objeto que contiene el parámetro a aleatorizar
- *property* → parámetro a aleatorizar utilizando una distribución normal
- *mean* → valor medio del parámetro
- *sd* → desviación estándar del parámetro

Se ha implementado una tercera función para aleatorizar parámetros booleanos:

`randomizeBool(`*source, property, probability*`)`

- *source* → objeto que contiene el parámetro a aleatorizar
- *property* → parámetro a aleatorizar
- *probability* → probabilidad de que el parámetro tome valor *true*

### 6.2.1. Distribución uniforme

La **distribución uniforme** es una distribución de probabilidad que se caracteriza porque todos los valores en un intervalo $[min, max]$ son **equiprobables**.

La función `Math.random()` de Javascript genera valores pseudo-aleatorios distribuidos uniformemente en el rango $[0,1)$, que se pueden convertir al rango $[min, max)$ aplicando la siguiente transformación:

$$y = x(max - min) + min$$

### 6.2.2. Distribución normal

La **distribución normal** (también llamada distribución gaussiana), es una distribución de probabilidad que se usa frecuentemente en el modelado de fenómenos naturales. La altura y el peso de un individuo son dos ejemplos de variables que responden a este tipo de distribución.

Para generar valores que sigan una distribución normal se han probado dos métodos aproximativos distintos:

- Mediante la suma de $n$ variables aleatorias distribuidas uniformemente
- Mediante la transformación de Box-Muller

El primer método se sustenta en el **teorema del límite central**. Este teorema establece que siendo $Sn$ la suma de $n$ variables aleatorias independientes idénticamente distribuidas, entonces la distribución de $Sn$ tiende a una distribución normal a medida que $n$ aumenta.

Si las variables aleatorias sumadas están distribuidas uniformemente en el intervalo $[0,1]$ entonces la distribución resultante recibe el nombre de **distribución de Irwin-Hall**. La media de la distribución será $\mu = \frac{n}{2}$ y su varianza $\sigma^2 = \frac{n}{12}$.

El segundo método utilizado para aproximar una distribución normal ha sido **la transformación de Box-Muller** [Box and Muller, 1958]. Esta transformación toma dos variables independientes $U_1$, $U_2$ distribuidas uniformemente en el intervalo $(0,1)$ y genera dos variables $X_1$, $X_2$ que siguen una distribución normal con $\mu = 0$ y $\sigma^2 = 1$. La relación entre las variables es la siguiente:

$$X_1 = \sqrt{-2\ln U_1}\cos(2\pi U_2)$$

$$X_2 = \sqrt{-2\ln U_1}\sin(2\pi U_2)$$

Independientemente del método utilizado, una vez obtenida una distribución normal estándar $Z \sim N(0,1)$ podemos transformarla en otra distribución $N(\mu,\sigma^2)$ aprovechando la siguiente relación:

$$N = \sigma Z + \mu$$

Por motivos de eficiencia se ha escogido el método de Box-Muller, ya que el otro método requiere muchas muestras para conseguir una buena aproximación de la distribución normal.

## 6.3. Relaciones entre parámetros

Además de aleatorizar el valor de ciertos parámetros, en algunos casos también nos puede interesar establecer relaciones entre ellos. Por ejemplo, relacionando la altura del caballo con su peso podríamos conseguir que las variaciones más altas sean también las más pesadas. Esta funcionalidad está implementada en la siguiente función:

`bindProperty (source, sourceProperty, target, targetProperty, func)`

- *source* → objecto que contiene el parámetro de origen
- *sourceProperty* → parámetro de origen
- *target* → objeto que contiene el parámetro a relacionar
- *targetProperty* → parámetro a relacionar
- *func* → función que describe la relación entre ambos parámetros

## 6.4. Fuentes para definir parámetros

El principal objetivo que se persigue en este proyecto es tratar de trasladar al mundo virtual la variedad que presentan los objetos en la realidad.

Para que esto sea posible, es necesario que el comportamiento de los parámetros del sistema se asemeje al de sus equivalentes en el mundo real. Ello implica tener un grado de conocimiento sobre el modelo que en algunos casos puede no estar a nuestro alcance.

Afortunadamente, en el caso de los caballos uno puede encontrar gran cantidad de información en la literatura. Concretamente, el libro *"Simon & Schuster's Guide to Horses and Ponies"* [Bongianni, 1988] contiene información detallada acerca de las características físicas y del pelaje de más de 170 razas de caballos.

De esta información tiene especial interés para el proyecto la relativa al rango de valores que toman la altura y el peso de los caballos, así como la descripción de los distintos elementos que pueden estar presentes en el pelaje en función de la raza.

Incorporar toda la información contenida en el libro va más allá de las posibilidades de este proyecto, pero a pesar de ello se ha querido poner en valor la importancia de los datos empíricos en la creación de un sistema que genere resultados realistas. En el capítulo 8 de esta memoria se describe qué aspectos del libro se han incorporado en la demo final.

# 7. CAPÍTULO

# Integración en Unity

La última fase de este proyecto ha consistido en **integrar las clases presentadas en capítulos anteriores en Unity**, un motor gráfico multiplataforma orientado al desarrollo de videojuegos. Este capítulo describe los pasos que se han dado para conseguirlo.

## 7.1. Aprendizaje

El primer paso para integrar el sistema en Unity ha consistido en **familiarizarse con la herramienta**. Uno de los aspectos positivos de Unity es la cantidad de información que hay disponible en su propia web, que contiene una sección entera dedicada al aprendizaje. Entre los recursos que se presentan en dicha sección destacan los tutoriales, que sirven de introducción a las principales funcionalidades del motor.

La primera toma de contacto con Unity ha consistido en realizar el tutorial *"Roll-a-ball"*, en el que se enseñan aspectos básicos como el uso del editor, la creación de scripts o el manejo de objetos. El inconveniente de estos tutoriales es que están enfocados al desarrollo de videojuegos, y muchos de los temas que se tratan son de escaso interés para este proyecto. Por ello, tras haber finalizado este primer tutorial se ha procedido a empezar a implementar el sistema.

La principal fuente de información en la fase de implementación ha sido el manual de scripting de Unity [Unity Technologies, 2017].

## 7.2. Scripts

Los scripts son componentes que permiten introducir funcionalidades adicionales en las aplicaciones creadas en Unity.

Unity soporta dos lenguajes de programación para escribir scripts: C# y Javascript. La implementación presentada en capítulos anteriores está escrita en Javascript, lo cual posibilita la reutilización de gran parte del código. A pesar de ello, **se ha decidido utilizar C#** para aprovechar la oportunidad de aprender un nuevo lenguaje de programación.

### 7.2.1. Implementación del sistema

La implementación del sistema en Unity ha consistido en la creación de varios scripts, uno por cada componente del sistema:

- **Componentes principales**: `ModelCustomizer`, `Modifier`, `TextureModifier`, `SkeletonModifier`, `ProceduralTexture`
- **Componentes auxiliares**: `Utilites`, `GraphicUtilities`, `TextureBlender`

Desde el punto de vista de la funcionalidad, la implementación realizada en Unity es equivalente a la implementación original. En la figura 7.1 se muestran algunos de los resultados obtenidos en Unity.

**Figura 7.1:** Modelo original (izquierda) junto con dos variaciones generadas en Unity

Donde sí existen diferencias es en ciertos aspectos de la implementación, concretamente, en el procesamiento del modelo y en el renderizado de texturas procedurales.

### 7.2.2. Procesamiento del modelo

Cuando el sistema recibe un modelo el primer paso que realiza es extraer su material, su textura y su esqueleto (para posteriormente aplicarles los correspondientes modificadores). Para poder realizar este procesamiento es necesario conocer la manera en la que Unity representa los objetos.

En Unity todos los elementos de una escena (luces, cámaras, modelos...) son instancias de la clase `GameObject`. Un `GameObject` puede contener varios componentes, que son los que dotan de funcionalidad al objeto. El modelo 3D utilizado en este proyecto tiene los siguientes componentes (Figura 7.2):

- **Transform**: contiene la posición, rotación y escala del modelo.
- **Skinned Mesh Renderer**: contiene la información necesaria para el renderizado del modelo (la malla, los materiales, el esqueleto...).
- **Material**: contiene el *shader* utilizado para renderizar el modelo y la información que dicho *shader* necesita (texturas, parámetros...).

Por tanto, el sistema puede acceder a la información que necesita a través del componente `Skinned Mesh Renderer` del objeto.

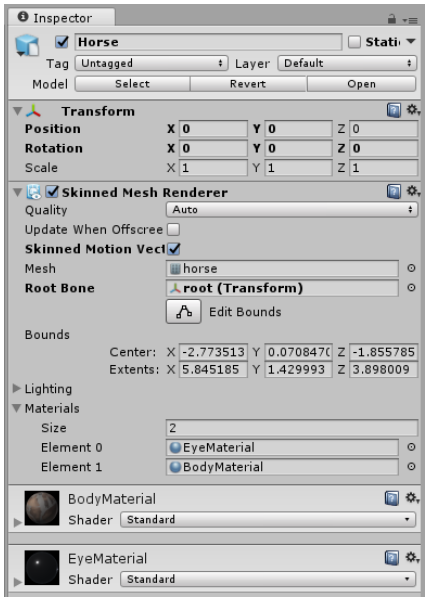



**Figura 7.2:** Componentes de Unity asociados al modelo 3D utilizado

### 7.2.3. Renderizado de texturas

Para renderizar una textura procedural se necesitan tres elementos: la malla del modelo (para el mapeado UV), el *shader* que genera la textura, y una textura en la que almacenar el resultado. En Unity las texturas utilizadas para almacenar el resultado de un renderizado son de la clase `RenderTexture`.

El proceso de renderizado consta de los siguientes pasos:

1. Crear un material que contenga el *shader* y los valores paramétricos de la textura procedural.
2. Crear una `RenderTexture` e indicarle al motor gráfico que guarde en ella el resultado del próximo renderizado.
3. Renderizar la malla del modelo con el material creado en el primer paso.

Para realizar estos pasos desde un script ha sido necesario utilizar las siguientes funciones de la API de Unity: `Graphics.SetRenderTarget`, `Graphics.DrawMeshNow` y `Graphics.Blit`.

La especificación completa de dichas funciones está disponible en el manual de scripting de Unity [Unity Technologies, 2017].

## 7.3. Shaders

Los *shaders* de la implementación original del sistema están escrito en *glsl*, el lenguaje de *shading* de OpenGL y WebGL. Aunque Unity permite utilizar *shaders* escritos en este lenguaje, la recomendación es escribirlos en *hlsl*. Esta recomendación es debida a que Unity es capaz de traducir los *shaders* escritos en *hlsl* a otros lenguajes (*glsl* incluido), lo cual permite ejecutar la aplicación en muchas más plataformas.

La traducción de un lenguaje a otro ha sido una tarea sencilla, que ha consistido principalmente en cambiar el tipo de algunas variables y los nombres de varias funciones. En *"GLSL-to-HLSL reference"* [Microsoft, 2017] se describen de manera concisa las principales diferencias entre ambos lenguajes.

# 8. CAPÍTULO

# Integración y demostración

En este capítulo se presenta una visión global del trabajo realizado, que abarca tanto el uso práctico del sistema como una demostración de los resultados que pueden obtenerse.

## 8.1. Uso práctico del sistema

Dado que el sistema carece de interfaz de usuario, para poder utilizarlo es necesario trabajar a nivel de código. El requisito de partida es disponer de un modelo 3D que contenga una malla, una textura y un esqueleto. Para generar variaciones de dicho modelo el usuario tiene que definir un conjunto de modificadores y asignarlos a un objeto de la clase `ModelCustomizer`, desarrollada en Javascript para este proyecto.

Esquema general:

```
let customizer = new ModelCustomizer();

customizer.addModifier(modifier1);
customizer.addModifier(modifier2);
customizer.addModifier(...);
customizer.addModifier(modifierN);

customizer.customize(model);
```

El método `customize()` aplica los modificadores sobre el modelo indicado, dando lugar a una variación del mismo. En las siguientes secciones se describe la manera en la que se definen estos modificadores.

### 8.1.1. Modificadores de texturas

Para generar variaciones de la textura del modelo se utilizan modificadores de la clase `TextureModifier`. El usuario puede controlar el efecto de estos modificadores manipulando los valores de *lightness* y *saturation* (ver capítulo 2). A continuación se muestran algunos ejemplos de cómo utilizar estos modificadores para obtener variaciones del modelo de un caballo. El código que se presenta corresponde a la implementación en Javascript del sistema.

#### Ejemplo 1 - Generación de un caballo blanco

El siguiente modificador transforma la textura del modelo original en otra que se asemeja a la de un caballo blanco. Como los valores de *lightness* y *saturation* son fijos, el resultado obtenido es siempre el mismo.

```
let hairColor = new TextureModifier({
  lightness: 1.0;
  saturation: -1.0;
});
```

#### Ejemplo 2 - Generación un caballo de cualquier color

Si en vez de un caballo blanco quisiéramos que el sistema genere caballos de cualquier color, bastaría con utilizar un modificador cuyos valores de *lightness* y *saturation* sean aleatorios (ver capítulo 6):

```
let hairColor = new TextureModifier();

Utilities.randomizeFloat(hairColor, 'lightness', -1.0, 1.0);
Utilities.randomizeFloat(hairColor, 'saturation', -1.0, 0.0);
```

Utilizando este único modificador podemos crear múltiples variaciones del caballo que se diferencian en el color del pelaje (Figura 8.1):

```
let customizer = new ModelCustomizer();
customizer.addModifier(hairColor);
customizer.customize(model);
```

**Figura 8.1:** Variaciones generadas por el sistema utilizando un único modificador que aleatoriza el color del pelaje del caballo

### Ejemplo 3 - Generación de un caballo con manchas

El siguiente paso para crear variaciones más interesantes es utilizar texturas procedurales. Estas texturas actúan como máscaras y su aspecto viene determinado por los siguientes parámetros: *frequency, displacement, threshold, sharpness* y *distortion* (ver capítulo 3).

El siguiente modificador utiliza una textura procedural para añadir manchas al pelaje del caballo (Figura 8.2):

```
let spots = new ProceduralTexture({
  geometry: model.geometry, // Necesario para el mapeado UV
  frequency: 1.75,
  threshold: 0.85,
  sharpness: 10,
  max_z: 3.8,
});

Utilities.randomizeFloat(spots, 'displacement');

let bodySpots = new TextureModifier({
  lightness: 1.0;
  saturation: -1.0;
  masks: [spots]
});
```

Aleatorizando el valor de *displacement* conseguimos que las manchas del pelaje cambien de posición cada vez que se genera una nueva variación. También podemos introducir variabilidad en el tamaño de las manchas aleatorizando los valores de *frequency* y *threshold*.

Podemos controlar en qué partes del modelo se aplica la textura procedural asignando valores a *max_x, max_y, max_z, min_x, min_y* y *min_z* (ver capítulo 3). En el ejemplo

propuesto se utiliza esta opción para evitar que surjan manchas en la cabeza del caballo.

Finalmente, también cabe la posibilidad de cambiar la probabilidad de activación de un modificador. Por ejemplo, para conseguir que las manchas aparezcan únicamente en el 25 % de las variaciones bastaría con añadir la siguiente instrucción al ejemplo anterior:

```
16    Utilities.randomizeBool(bodySpots, 'enabled', 0.25);
```

**Figura 8.2:** Resultado obtenido al aplicar un modificador que utiliza una textura procedural para añadir manchas al pelaje de un caballo

### 8.1.2. Modificadores de mallas

Los modificadores utilizados para generar variaciones de la malla del modelo son de la clase `SkeletonModifier`. Estos modificadores alteran la forma del modelo aplicando transformaciones a su esqueleto (ver capítulo 5).

#### Ejemplo 4 - Generación de un caballo con tamaño de cola variable

Para cambiar el tamaño de la cola del caballo se realiza un escalado en el eje $z$ de los huesos *tail.001* y *tail.002*. La aleatorización del factor de escalado posibilita que el tamaño de la cola cambie de una variación a otra.

```
1  let tailModifier = new SkeletonModifier();
2  let scale = new THREE.Vector3(1.0, 1.0, 1.0);
```

```
3
4   Utilities.randomizeFloat(scale, 'z', 0.8, 1.2);
5
6   heightModifier.scaleBone('tail.001', scale);
7   heightModifier.scaleBone('tail.002', scale);
```

## 8.2. Demo final

### 8.2.1. Descripción de la demo

**Enlace a la demo**: http://isantesteban.com/tfg/demo4/

La demo final consiste en una única escena en la que se presenta un desfile de caballos. Cada vez que un caballo abandona la escena uno nuevo aparece, dando lugar a una marcha que nunca termina.

En la práctica la escena está compuesta por nueve réplicas de un mismo modelo. Cada una de estas réplicas es procesada por el sistema para conseguir que cada caballo sea diferente de los demás. Cuando un caballo alcanza el límite de la escena se realizan dos acciones:

1. Generar una nueva variación del modelo
2. Trasladar el modelo a la posición de inicio

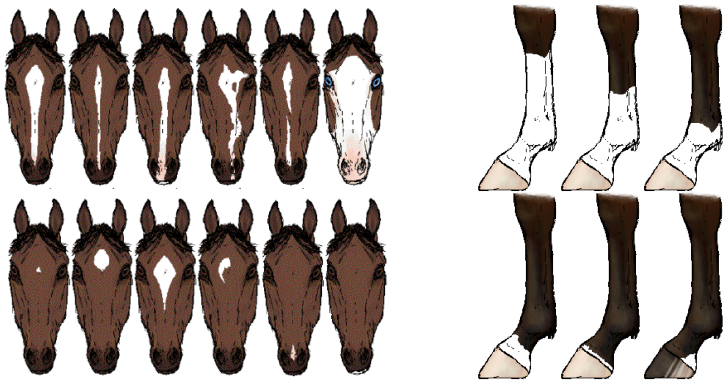
De este modo se consigue crear la ilusión de que cada caballo que entra en la escena es un nuevo modelo.

#### Generación de variaciones

La estrategia seguida para generar variaciones ha consistido en dividir el sistema en múltiples subsistemas. Cada subsistema se encarga de la generación de caballos con un tipo particular de pelaje: *bay*, *white*, *grey* o *dun*.

El libro *"Simon & Schuster's Guide to Horses and Ponies"* [Bongianni, 1988] describe cada uno de estos pelajes e incluye múltiples fotografías de referencia. Además del color de base, también describe otras características que pueden afectar al pelaje de un caballo, como manchas (Figura 8.3), marcas (Figura 8.4) y oscurecimiento en las extremidades (Figura 8.5).

**Figura 8.3:** Algunos ejemplos de las manchas presentes en el pelaje de algunos caballos (Fuente: *Wikipedia, La enciclopedia libre*)



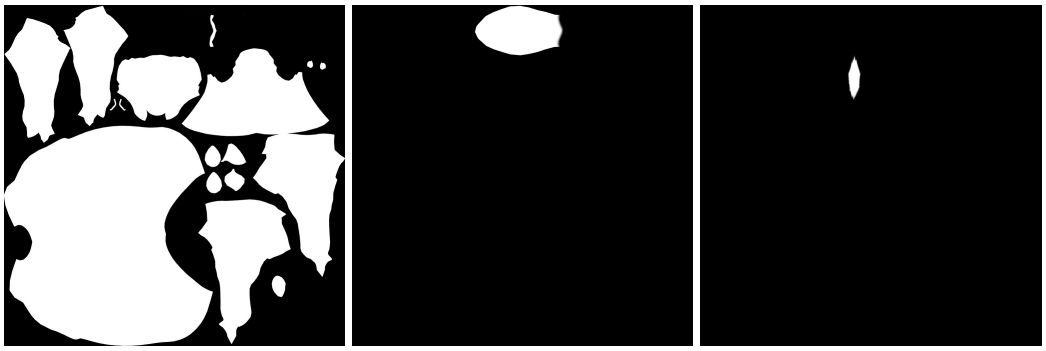
**(a)** Marcas faciales **(b)** Marcas en las patas

**Figura 8.4:** Ilustraciones que muestran el tipo de marcas presentes en el pelaje de caballos (Fuente: *Wikipedia, La enciclopedia libre*)

**Figura 8.5:** Algunos ejemplos de caballos que presentan oscurecimiento en las extremidades (Fuente: *Wikimedia Commons*)

Estas características del pelaje se han recreado utilizando modificadores de texturas de forma similar a los ejemplos presentados en la sección 8.1. Además de texturas procedurales, también se han utilizado tres máscaras auxiliares (generadas manualmente) para identificar las siguientes partes de la textura:

- Las partes correspondientes al pelaje (Figura 8.6a)
- La parte correspondiente a la cola (Figura 8.6b)
- La parte correspondiente a la frente (Figura 8.6c)



**(a)** Pelaje **(b)** Cola **(c)** Frente

**Figura 8.6:** Máscaras auxiliares utilizadas en la demo

La configuración completa del sistema se puede consultar en el fichero *HorseVariations.js*, que contiene la definición de todos los modificadores y texturas procedurales utilizados en esta demo.

### Movimiento de los caballos

Para conseguir que el movimiento de los caballos sea realista ha sido necesario crear manualmente una animación. Esta animación consiste en un ciclo que trata de capturar el paso de un caballo.

Para subsanar la falta de conocimiento previo en el campo de la animación, se ha buscado información en varias fuentes. Las dos que han resultado de mayor utilidad han sido el manual de referencia de Blender [Blender Documentation Team, 2016] y el libro *"The Animator's Survival Kit"* [Williams, 2012]. Este segundo incluye una explicación de cómo se mueven las distintas partes del caballo (patas, cabeza, cuerpo...) a lo largo de la animación.

El resultado final es una animación compuesta por 130 fotogramas que describe el movimiento de 25 huesos del caballo.

### 8.2.2. Resultados obtenidos

Esta demo pone de manifiesto el cumplimiento de los objetivos del proyecto:

- El sistema implementado es capaz de generar variaciones de la textura y de la forma del modelo utilizado.
- Las variaciones se generan automáticamente y en cuestión de milisegundos.

También se demuestra que las variaciones generadas pueden animarse sin inconvenientes, ya que la animación se ejecuta correctamente a pesar de las modificaciones realizadas en la forma del modelo.

Otro punto a destacar es el de la gestión de los recursos del sistema. Esta demo tiene la particularidad de que se generan variaciones sin parar, y cada variación introduce una nueva textura en memoria. Ello puede dar lugar a que se produzca una fuga de memoria y que se aborte la ejecución del programa si ésta se agota.

Afortunadamente, el sistema gestiona esta situación de forma adecuada, liberando de memoria aquellas texturas que no están siendo utilizadas. Para comprobar que el funcionamiento es el esperado se ha monitorizado el consumo de memoria de la GPU. Los resultados obtenidos han sido los siguientes:

- Consumo de memoria (variaciones desactivadas): 197MB
- Consumo de memoria (variaciones activadas): 352MB

En ambos casos el uso de memoria se mantiene constante, por lo que podemos concluir que no se produce ninguna fuga de memoria. En el primer caso el consumo es menor porque la escena está compuesta por un único modelo y una única textura (replicados nueve veces). Al activar la generación de variaciones el uso de memoria aumenta 155MB debido a que cada variación tiene su propia textura. Este coste podría reducirse considerablemente comprimiendo las texturas generadas.

Finalmente, mencionar que para realizar estas mediciones no se han tenido en consideración las texturas que el gestor de recursos mantiene en reserva para mejorar el rendimiento (ver sección 4.2.2).

# 9. CAPÍTULO

## Contribuciones

En este proyecto se ha desarrollado un **sistema que permite generar variaciones del modelo de un caballo**. Las variaciones se generan automáticamente a partir de un conjunto de modificadores definidos por el usuario, los cuales pueden afectar a la textura o a la forma del modelo. El sistema se ha implementado en dos entornos distintos:

- **Aplicaciones creadas con WebGL**. Una de las principales limitaciones de los gráficos 3D en la web es el tamaño que ocupan las aplicaciones, ya que repercute directamente en el tiempo que tarda el usuario en acceder al contenido.

  Generalmente esto lleva a diseñar escenas que utilicen el menor número posible de modelos, lo cual compromete el realismo de las mismas. Gracias al sistema implementado es posible crear escenas con un elevado nivel de diversidad sin que ello suponga un aumento del tamaño que ocupa la aplicación.

- **Aplicaciones desarrolladas en Unity**. El principal valor que ofrece el sistema en este caso es la posibilidad de conseguir gran variabilidad sin necesidad de generar manualmente cada variación.

La creación de una animación (caballo avanzando al paso) no formaba parte de los objetivos del proyecto, pero se ha realizado para exponer de un modo más realista e intuitivo sus contribuciones.

Por otro lado, aunque este proyecto se haya centrado en el modelo de un caballo, el sistema se ha diseñado con la intención de que se pueda **extender su uso a otros tipos de**

**modelos**. En el caso de otros cuadrúpedos la aplicación del sistema es trivial, aunque para obtener resultados convincentes sería necesario contar con información análoga para estos animales. Es posible que existan publicaciones similares a la de *"Simon & Schuster's Guide to Horses and Ponies"* [Bongianni, 1988] para otros cuadrúpedos. Este libro ha sido muy útil para ajustar de un modo realista los parámetros del caballo.

La generación de variaciones de otros tipos de modelos (que no sean animales) requeriría ampliar la variedad de modificadores y texturas procedurales que se pueden utilizar. Gracias a la **modularidad del sistema** incorporar nuevas funcionalidades es una tarea sencilla.

En resumen, el sistema desarrollado **ha satisfecho las expectativas del proyecto** y está abierto a trabajos futuros que amplíen sus capacidades.

# Bibliografía


[Bao et al., 2013] Bao, F., Schwarz, M., and Wonka, P. (2013). Procedural facade variations from a single layout. *ACM Transactions on Graphics (TOG)*, 32(1).

[Biagioli, 2014] Biagioli, A. (2014). Understanding perlin noise. http://flafla2.github.io/2014/08/09/perlinnoise.html. (Fecha acceso: 2017-01-02).

[Blender Documentation Team, 2016] Blender Documentation Team (2016). Blender reference manual. https://docs.blender.org/manual/en/dev/. (Versión: 2.78).

[Bongianni, 1988] Bongianni, M. (1988). *Simon & Schuster's Guide to Horses and Ponies*. Fireside Books.

[Box and Muller, 1958] Box, G. E. P. and Muller, M. E. (1958). A note on the generation of random normal deviates. *Annals of Mathematical Statistics*, 29(2).

[Contreras, 2016] Contreras, C. (2016). Horse rig. http://www.mothman-td.com/portfolio_items/horse_rig/. (Fecha acceso: 2016-11-21).

[Ebert et al., 2002] Ebert, D. S., Musgrave, F. K., Peachey, D., Perlin, K., and Worley, S. (2002). *Texturing and Modeling, Third Edition: A Procedural Approach*. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 3 edition.

[Gustavson, 2005] Gustavson, S. (2005). Simplex noise demystified. Linköping University, Sweden.

[Jain et al., 2012] Jain, A., Thormählen, T., Ritschel, T., and Seidel, H.-P. (2012). Exploring shape variations by 3d-model decomposition and part-based recombination. *Computer Graphics Forum*, 31(2.3).

[Microsoft, 2017] Microsoft (2017). GLSL-to-HLSL reference. https://docs.microsoft.com/en-us/windows/uwp/gaming/glsl-to-hlsl-reference. (Fecha acceso: 2017-04-17).

[Randima, 2004] Randima, F., editor (2004). *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*. Pearson Education, Inc.

[Stava et al., 2014] Stava, O., Pirk, S., Kratt, J., Chen, B., Mech, R., Deussen, O., and Benes, B. (2014). Inverse procedural modelling of trees. *Computer Graphics Forum*, 33(6).

[Turk, 1991] Turk, G. (1991). Generating textures on arbitrary surfaces using reaction-diffusion. *Computer Graphics*, 25(4):289–298.

[Unity Technologies, 2017] Unity Technologies (2017). Unity scripting reference. https://docs.unity3d.com/ScriptReference/index.html. (Versión: 5.6).

[Williams, 2012] Williams, R. (2012). *The Animator's Survival Kit: A Manual of Methods, Principles and Formulas for Classical, Computer, Games, Stop Motion and Internet Animators*. Farrar, Straus and Giroux.

[Øyvind Kolås, 2005] Øyvind Kolås (2005). Image processing. https://pippin.gimp.org/image-processing/. (Fecha acceso: 2016-12-22).