UNIVERSIDAD NACIONAL DEL LITORAL
FACULTAD DE INGENIERÍA Y CIENCIAS HÍDRICAS

Tecnicatura en diseño y programación de videojuegos

UNL VIRTUAL

Programación para videojuegos II

ANEXO 1 (2012)
Manejo de escenas

Docente
Pablo Abratte

Índice

1. Introducción

Al analizar el comportamiento de videojuegos, puede notarse que los mismos transcurren en distintas escenas o estados y se que mueven de manera transparente entre los mismos.

Tomemos como ejemplo un conocido clásico de las consolas *Sega Mega Drive/Genesis: Streets of Rage* (Fig. 1). Como la mayoría de los juegos para éstas consolas, comienza mostrando el logo de *Sega*. A continuación, sigue una pantalla donde se muestra una vista de la ciudad y el texto con la trama de la historia se desplaza de abajo hacia arriba, esta escena puede interrumpirse en cualquier momento presionando una tecla para proceder a la escena siguiente: la pantalla de portada, la cual muestra un dibujo de los personajes con la cuidad de fondo y un mensaje que pide al jugador que presione el botón *Start* para comenzar. Acto seguido, el jugador es llevado a la pantalla de menú principal, la cual exhibe un fondo de color negro y permite seleccionar entre el modo de 1 o 2 jugadores, o acceder a un nuevo menú para ajustar otras opciones del juego. Si decide comenzar, el jugador es llevado a una pantalla de selección en la que se muestran los personajes disponibles con sus respectivas habilidades para que escoja uno. Finalmente, el juego comienza. La pantalla principal del juego consiste en calles en las que los héroes pueden moverse de manera horizontal y hacia arriba y hacia abajo (este tipo de movimiento es generalmente conocido como *belt-scrolling*) y deben propinar golpes a delincuentes que encuentran en su camino. Al perder toda su energía y vidas, el jugador es llevado a una pantalla en la que se muestran los puntajes más altos obtenidos, para volver luego a la escena del logo de *Sega* y volver a comenzar el ciclo.

Figura 1: Capturas de pantalla de distintas escenas del videojuego *Streets of Rage*: pantalla de título (izquierda), selección de personaje (centro) y pantalla principal de juego (derecha).

Como se ilustra en el ejemplo anterior, los distintos estados de un juego suelen ser muy diferentes en cuanto a su lógica, dibujado y manejo de eventos. La diferencia es tal, que a menudo da la impresión de cada estado se tratase de un programa independiente y el juego, en su totalidad, resultáse de la unión de éstos subprogramas.

La manera tradicional y directa de manejar múltiples estados o escenas consiste en utilizar una serie de estructuras iterativas y condicionales dentro del bucle principal de

juego para realizar las acciones correspondientes a la escena actual. Este enfoque, sin embargo, no es recomendable, ya que complica el mantenimiento y depuración del código del juego.

En las siguientes secciones se propone una estrategia para manejar distintas escenas o estados de juego de manera simple, transparente y escalable.

2. Definición de escena

Como se mencionó anteriormente, las distintas escenas o estados de juego se comportan como pequeños juegos independientes, con sus propias formas diferenciadas de manera de manejar eventos, actualizar y dibujar. Es decir, poseen su propia implementación de cada una de las principales partes del bucle de juego.

La estrategia propuesta consiste en representar los distintos estados del juego mediante objetos, de manera que cada uno posea su propia definición de las funciones principales observadas en la Fig. 2.

De este modo, el bucle de juego se encargará de llamar a las funciones de manejo de eventos, actualización y dibujado correspondientes al objeto que represente a la escena actual. Los cambios de escena, como se verá más adelante, podrán realizarse de manera muy sencilla cambiando dicho objeto.

Entre las numerosas ventajas que éste enfoque provee, pueden citarse las siguientes:

- Permite un bucle principal de juego sencillo, pequeño y prolijo, y elimina la necesidad de realizar modificaciones sobre el mismo para incluir nuevas escenas.
- Se separa el código correspondiente a las implementaciones de los distintos estados de juego.
- Es posible crear escenas arbitrariamente complejas.
- Agregar nuevas escenas es sencillo y no requiere modificar las que ya fueron programadas.

Para representar a una escena, se propone el prototipo de clase observado en la Fig. 3.

Para crear nuevas escenas, se crearán clases que hereden de la clase abstracta **Scene** y se definirán sus funciones. El motor de juego, será el encargado de ejecutar

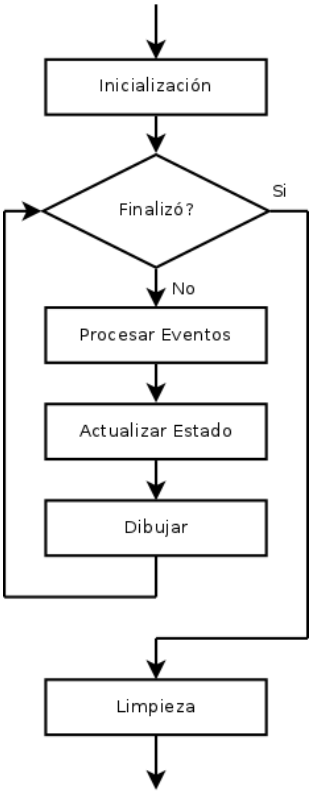

Figura 2: Esquema de un bucle de juego genérico.

```
1  class Scene {
2    public:
3    virtual void Init() = 0;
4    virtual void ProcessEvent(const sf::Event &e) = 0;
5    virtual void Update(const float &dt) = 0;
6    virtual void Render(const sf::RenderWindow &w) = 0;
7    virtual void Cleanup() = 0;
8  };
```

Figura 3: Prototipo de clase que representa una escena.

el gameloop y llamar a dichas funciones para cada escena.

En la clase propuesta, los métodos *Init(...)* y *Cleanup(...)* llevan a cabo la inicialización y limpieza de la escena. La existencia de éstos métodos agrega mayor flexibilidad a la utilización de éstos objetos, al permitir mayor control sobre la creación y destrucción de los mismos que el que permiten los constructores y destructores, sobre los cuales no se tiene control sobre su invocación. De esta manera, una escena puede crearse desde el primer momento en que el juego es cargado en memoria, pero su carga y liberación de recursos podrá llevarse a cabo cuando resulte conveniente, lo cual permitirá ganar eficiencia en el uso de recursos.

La función *ProcessEvent(...)* recibirá un evento de SFML y podrá modificar el estado de la escena en función del mismo.

La función *Update(...)* recibirá un valor real con el tiempo transcurrido desde la última actualización, a partir del cual actualizará el estado de la escena.

Finalmente, la función *Render(...)* recibe la ventana de dibujo para poder dibujar la escena.

3. El manejador de escenas

En esta sección se abordará el diseño de la clase encargada del manejo de escenas. Como se mencionó anteriormente, dentro del bucle de juego se deberá identificar al objeto que correspondiente a la escena actual y llamar a la funciones *ProcessEvent(...)*, *Update(...)* y *Render(...)* de dicho objeto, proporcionando los parámetros necesarios. Para crear nuevos tipos de escenas, el desarrollador deberá heredar de la clase **Scene** y redefinir sus métodos.

La clase encargada de correr el bucle de juego y del manejo de escenas se ha denominado **Engine** y su prototipo puede observarse en la Fig. 4.

Entre sus atributos, la clase posee la ventana de juego y dos punteros para hacer referencia a la escena actual y a la próxima. Éste último se utilizará en caso de un cambio de escena. Los métodos propuesto permiten inicializar el motor especificando el tamaño de la ventana de juego, ejecutar el bucle principal de juego y cambiar la escena actual.

La función *Loop(...)* es la que reviste mayor importancia, y su código puede observarse en la Fig. 5.

```
class Engine {
  private:
  sf::RenderWindow m_window;
  Scene *m_currentScene, *m_nextScene;

  public:
  void Init(unsigned w, unsigned h);
  void Loop();
  void ChangeScene(Scene *s);
};
```

Figura 4: Prototipo para la clase **Engine**, encargada del manejo de escenas.

```
void Engine::Loop(){
  bool exit = false;
  sf::Clock clock;
  float elapsedTime;
  sf::Event event;

  // el bucle principal del juego
  clock.Reset();
  while(!exit){
    // calcula el tiempo transcurrido
    elapsedTime=clock.GetElapsedTime();
    clock.Reset();

    // procesa los eventos
    while(m_window.GetEvent(event)) {
      m_currentScene->ProcessEvent(event);
    }

    // actualiza la escena actual
    m_currentScene->Update(elapsedTime);

    // dibuja la escena actual
    m_currentScene->Render(m_window);
    m_window.Display();

    // si es necesario realizar un cambio de escena
    if (m_nextScene != NULL){
      // la escena actual se desinicializa y borra
      m_currentScene->Cleanup();
      delete m_currentScene;
      // se inicializa la nueva escena
      m_currentScene = m_nextScene;
      m_currentScene->Init();
      m_nextScene = NULL;
    }
  }
}
```

Figura 5: Código de la función *Engine::Loop(...)*.

Las escenas definidas por el usuario serán clases heredadas de **Scene**. Dado que el tipo del puntero *m_currentScene* corresponde a dicha clase base, y que en la misma los métodos fueron declarados como virtuales, la invocación a métodos utilizando este puntero hará uso de los mecanismos de polimorfismo. Es decir, el puntero *m_currentScene* no apuntará a objetos de tipo **Scene** sino a clases heredadas de ésta última definidas por el usuario. El uso de *virtual* permitirá crear una tabla de métodos virtuales que servirá al programa para decidir, en tiempo de ejecución, a cuál de las clases de la jerarquía hace referencia el puntero.

Al final cada iteración del bucle de juego se realiza el manejo de escenas: si la dirección del puntero *m_nextScene* es distinta de *NULL*, entonces debe realizarse un cambio de escena. Dicho cambio consiste en liberar los recursos de la escena actual e inicializar la nueva escena, al tiempo que se hace que el puntero *m_currentScene* apunte a la misma.

Como es de suponer, el usuario puede realizar un cambio de escena invocando al método *ChangeScene(...)*, el cual modifica la dirección de *m_nextScene*.

En este punto cabe destacar que el cambio de escena es llevado a cabo únicamente por el motor. El usuario, mediante la invocación a *ChangeScene(...)*, sólo señalizará la intención de realizar dicho cambio, pero será el motor quien lo haga efectivo una vez que haya finalizado la iteración actual.

El hecho de que el control del bucle principal esté a cargo del motor y no del usuario es muy importante, ya que evita errores que podrían producirse si, durante una iteración, se actualizara una escena y se dibujara otra diferente, lo cual ocurriría al cambiar de escena en el momento inmediato a la solicitud del usuario y no al final de la iteración.

Finalmente, resta dedicar unas palabras a la forma en que se realizan los cambios de escena desde el punto de vista del programador. Éste debe realizar una primer llamada a *ChangeScene(...)* antes de entrar al bucle principal de juego para indicar la escena inicial. Además de eso, no existe ningún tipo de orden o secuencia entre las escenas, sino que cada una es responsable de conocer su predecesora e indicar el cambio hacia la misma. Es decir que cada escena, al finalizar, deberá invocar a *ChangeScene(...)* especificando la próxima escena.

El enfoque propuesto, a pesar de ser atractivo por su simpleza y ventajas, posee también limitaciones. En la siguiente sección se discutirán mejoras para flexibilizar y agregar funcionalidades al esquema.

4. Manejo avanzado de escenas

En esta sección se discutirán algunos inconvenientes del enfoque expuesto anteriormente y se propondrá un diseño avanzado, basado en el anterior, para manipular escenas de manera más flexible.

Como se mencionó, para realizar los cambios entre escenas, cada escena debe conocer su sucesora y especificar la transición a la misma mediante una llamada a *ChangeScene(...)*. Este hecho implica algunas desventajas que se mencionarán a continuación.

El primer lugar, existe una alta interdependencia entre las escenas: en cada escena estará codificada la creación de la escena sucesora, sin posibilidades de alterar, en tiempo de ejecución, el tipo de esta última. Este hecho puede verse agravado en el caso de escenas que puedan ser sucedidas por otras de distintos tipos según la situación, por ejemplo en caso que la pantalla de puntajes más altos pueda ser sucedida por la pantalla de título, cuando el jugador ha perdido, o por una pantalla de demostración si es que el usuario no comenzó el juego.

En segundo lugar, no existe la posibilidad de preestablecer de antemano una sucesión ordenada de escenas. Es decir, no puede crearse una lista de escenas de manera que cuando la actual finalice, la siguiente en la lista tome su lugar. Esta característica resulta muy útil en casos en los que el orden de determinadas escenas es conocido con anterioridad. Por ejemplo, en el caso de *Streets of Rage* mencionado anteriormente, la escena de logo es siempre seguida por la de introducción y ésta, a su vez, por la de título.

Finalmente, otra de las carencias del esquema es la falta de una funcionalidad que permita suspender una escena para pasar de manera temporal a otra y retomar la primera más tarde. Este es el caso de una escena de pausa en la que la escena de juego se deja suspendida hasta que la de pausa retorna.

Resumiendo, las funcionalidades que se desean agregar al nuevo manejador de escenas son las siguientes:

- Manipular una lista de escenas, de manera que la sucesión de las escenas pueda ser determinado el orden en dicha lista y pueda programarse escenas independientemente de su predecesora y sucesora.

- Agregar una escena dejando en suspenso la escena actual para poder, cuando la escena agregada finalice, retomar la primera en el mismo estado que fue dejada.

La primera de estas funcionalidades puede lograrse utilizando una estructura de datos como una lista o arreglo que almacene escenas, de manera que puedan agregarse escenas al final de la misma y se quiten desde el principio. Así, las escenas se ejecutarán en el orden que fueron agregadas, como en una estructura de tipo cola (FIFO).

Para la segunda funcionalidad mencionada, puede tratarse a la estructura de datos como una pila (LIFO): cuando se agregue una escena, la que estaba en ejecución se mantiene en memoria y será retomada cuando la escena apilada sobre ella sea quitada.

El tratamiento propuesto consiste en utilizar una cola de doble extremo, esta estructura de datos es similar a una cola pero permite inserciones y remociones en ambos extremos. En la Fig. 6 se ilustra la modalidad propuesta. El programador podrá agregar nuevas escenas en el extremo *B* de la cola, el motor ejecutará la escena en el extremo opuesto y, cuando ésta finalice, la quitará y seguirá con la siguiente. También es posible insertar una escena en el extremo *A*, en este caso la escena que estaba ejecutándose quedará en su estado actual y comenzará a ejecutarse la escena apilada. Cuando esta última finalice, será quitada de la cola y la ejecución continuará con la escena siguiente como se describió antes.

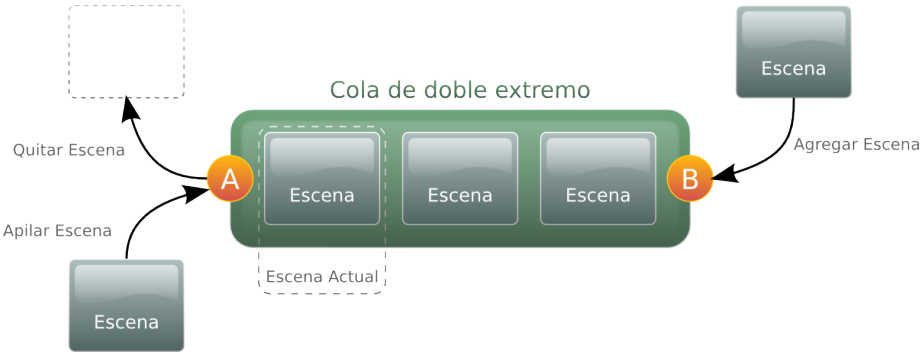

Figura 6: Almacenamiento y manipulación de escenas mediante una cola de doble extremo.

En el lenguaje C++, la biblioteca STL posee una implementación de cola de doble extremo denominada **deque** (double-ended queue). Para una referencia sobre las funciones disponibles en dicha estructura puede consultarse ésta página.

En la Fig. 7 puede observarse el nuevo prototipo para la clase **Engine**. Se han agregado las funciones *AppendScene(...)* y *PushScene(...)* para agregar una escena en el extremo *B* o *A* de la cola respectivamente, mientras que las funciones *PopScene(...)* y *EndScene(...)* permiten finalizar la escena actual y proseguir a la siguiente.

El atributo *m_sceneQueue* corresponde a la cola de doble extremo donde se alma-

cenarán punteros a las escenas y la bandera *m_mustUpdateCurrentScene* indica si debe realizarse algún tipo de operación de actualización con la escena actual.

Finalmente, los punteros *m_currentScene* y *m_sceneToDelete* apuntaran a la escena actual y a la escena que deberá ser borrada al quitarse de la cola, en caso de haber alguna.

```
1  class Engine {
2    public:
3    void Init(unsigned w, unsigned h);
4    void Loop();
5
6    void ChangeScene(Scene *s);
7    void PushScene(Scene *s);
8    void AppendScene(Scene *s);
9    void PopScene();
10   void EndCurrentScene();
11
12   private:
13   sf::RenderWindow m_window;
14   std::deque<Scene *> m_sceneQueue;
15   Scene *m_currentScene, *m_sceneToDelete;
16   bool m_mustUpdateCurrentScene;
17 };
```

Figura 7: Prototipo para la clase **Engine** mejorada.

El funcionamiento de las funciones para la manipulación del flujo de escenas es bastante sencillo. En la Fig. 8 se expone, a modo de ejemplo, el código de la función *PushScene(...)*. En la misma se inicializa la escena pasada como parámetro y se la inserta en el frente de la cola. Además, se indica que debe actualizarse el puntero a la escena actual y que no hay ninguna escena que deba ser desinicializada y borrada.

```
1  void Engine::PushScene(Scene *s){
2    s->Init();
3    m_sceneQueue.push_front(s);
4    m_mustUpdateCurrentScene = true;
5    m_sceneToDelete = NULL;
6  }
```

Figura 8: Código de la función *Engine::PushScene(...)*.

El código modificado de la función *Loop(...)* puede observarse en la Fig. 9. En éste caso, si es necesario actualizar el puntero a la escena actual, se hace tomando la próxima escena en el frente de la cola. También se borra, en caso de ser necesario, la escena que finalizada apuntada por *m_sceneToDelete*.

```
void Engine::Loop(){
  sf::Clock clock;
  float elapsedTime;
  sf::Event event;

  // el bucle de juego
  clock.Reset();
  while(!exitEngine){
    elapsedTime=clock.GetElapsedTime();
    clock.Reset();

    // procesa los eventos y llama a la funcion de eventos de la escena
    while(m_window.GetEvent(event)) {
      m_currentScene->ProcessEvent(event);
    }

    // actualiza la escena
    m_currentScene->Update(elapsedTime);

    // dibuja la escena
    m_currentScene->Render(m_window);
    m_window.Display();

    // manejo de escenas: si hubo algun cambio de escena, lo hace efectivo
    if (m_mustUpdateCurrentScene){
      // si hay alguna escena por borrar/desinicializar, lo hace
      if (m_sceneToDelete != NULL){
        m_sceneToDelete->Cleanup();
        delete m_sceneToDelete;
        m_sceneToDelete = NULL;
      }
      // pone como escena actual la escena del frente de la cola
      m_currentScene = m_sceneQueue.front();
    }
  }
}
```

Figura 9: Código de la función *Engine::Loop(...)*.

5. Transiciones entre escenas

El manejador de escenas presentado anteriormente implementa las transiciones entre escenas de manera que estas ocurren de manera inmediata. De esta manera, en una iteración se actualiza y dibuja una escena y, si se dió la orden de cambio de escena al motor, en la próxima iteración se actualizará y dibujará otra distinta. En muchos videojuegos, sin embargo, dichas transiciones ocurren de manera gradual y acompañadas generalmente de efectos gráficos fundidos graduales , pixelados de las imágenes (como en el caso del videojuego *Super Mario World* de la consola *Super Nintendo*), blur, bloom, etc. En esta sección, si bien no se presentarán detalles de implementación, se explicará como se puede hacer uso del enfoque presentado en el documento para realizar este tipo de transiciones de manera sencilla.

Tomándo como base la arquitectura de escenas propuesta, este tipo de transiciones pueden definirse como un tipo especial de escena, la cual que conoce y controla las escenas saliente y entrante. De este modo, la transición es responsable de la actualización y dibujado de ambas escenas, pero puede hacerlo utilizando su propia lógica.

Propongamos una clase de transición de ejemplo: la misma poseerá como atributos dos punteros a objetos de tipo **Scene** (la escena anterior y la próxima). En la implementación de su función *Update(...)* calculará el tiempo total que ha transcurrido desde que comenzó la transición y llamará a los métodos *Update(...)* de ambas escenas. El método *Draw(...)* podrá dibujará ambas escenas pero cambiando los valores de transparencia para cada una basándose en el tiempo transcurrido calculado anteriormente. De esta manera, mientras dure la transición, las escenas se dibujarán de manera que una aparezca de forma gradual al mismo tiempo que la otra desaparezca. Una vez finalizada la transición, se deberá descartar tanto la escena saliente como la de transición, al tiempo que la próxima escena se convierte en la actual.

También deberán realizarse algunas modificaciones menores sobre el motor para que pueda manejar correctamente este tipo especial de escenas. Las funciones *ChangeScene(...)*, *PushScene(...)*, etc, podrán recibir como parámetro opcional el tipo de transición a utilizar, el motor entonces se encargará de crear la transición a partir de las escenas correspondientes y asignarla como escena actual.

Si bien la actualización simultánea de las escenas involucradas en la transición no supone dificultades, el renderizado merece algunos comentarios adicionales. Para poseer flexibilidad en el dibujado de la transición, lo ideal es que cada escena sea renderizada en una textura, para luego utilizar ambas texturas en sprites con distintas propiedades para componer la transición. Como ejemplo, puede dibujarse ambas escenas como sprites que se desplazan hacia adentro y afuera de la pantalla variando su tamaño y transparencia.

La versión 1.6 de *SFML* no cuenta con funcionalidades para renderizar sobre textura. Sin embargo, los mismos resultados pueden lograrse con un pequeño truco: puede obtenerse una textura con una captura de pantalla utilizando el método *CopyScreen(...)* de la clase **sf::Image**, la misma recibe como parámetro la ventana de renderizado y copia los contenidos de la misma en el objeto que hace la llamada. Haciendo uso de esta funcionalidad, el renderizado de una transición puede llevarse a cabo de la siguiente manera:

1. Se limpia la pantalla y se dibuja la primer escena.

2. Se toma una captura de la pantalla y se almacena en la primer textura.

3. Se limpia la pantalla y se dibuja la segunda escena.

4. Se toma una captura de la pantalla y se almacena en la segunda textura.

5. Se limpia nuevamente la pantalla y se dibuja la transición haciendo uso de las texturas adquiridas anteriormente.

6. Bibliografía

- **Managing Game States in C++ [en línea]** `http://gamedevgeek.com/tutorials/managing-game-states-in-c/`