II. PROYECTOS REALIZADOS

A continuación se presentarán los informes de las actividades realizadas para los proyectos referentes a la plataforma de Second Life y a la de Unity.

II.1. SECOND LIFE

II.1.1. Turbina Pelton

La primera actividad a seguir durante la fase de trabajo realizado en Second Life fue la creación virtual de una Turbina Pelton, misma que se llevó a cabo en una semana.

Una turbina Pelton es una turbina hidráulica; una turbo-máquina motora de flujo trasversal, admisión parcial y acción. Debe su nombre a Lester Allen Pelton, quien concibió la idea de unas cucharas periféricas que aprovecharan la energía cinética del agua que venía de una tubería y actuaba tangencialmente sobre la rueda.

Debido a esto, se propuso como objetivo construir el modelo virtual de la turbina Pelton y añadirle un script para animarla como si en verdad estuviera aprovechando la energía cinética del agua.

La turbina Pelton fue un proyecto perfecto para iniciar con el modelado geométrico en el ambiente virtual de Second Life. Se utilizó como base un cilindro. La turbina en sí está compuesta por dicho cilindro y una serie de esferas medias con un agujero. No se utilizó ninguna textura para dar el efecto metálico; sólo se alteró el color especular y se eligió un brillo de intensidad media. El resto de la turbina fue modelado usando el video contenido en la siguiente liga como referencia: http://www.youtube.com/watch?v=HzQPNpP55xQ.

Una vez terminado el modelo, un colega insertó el código que le permitía a la turbina desplegar un comportamiento “funcional”, mismo que mostraba a las cucharas ser “golpeadas” por partículas de agua para así generar partículas de electricidad. Adicionalmente, se agregó un botón de encendido/apagado para iniciar y finalizar la animación.

Una presa con animación fue incluida por compañeros para complementar el modelo.

A continuación se muestra una imagen de la turbina Pelton finalizada:

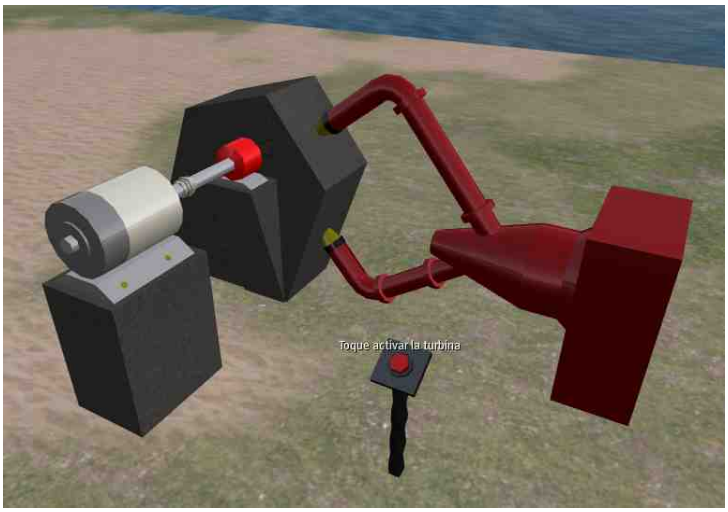


Fig. 2.1: Turbina Pelton en Second Life.

Este modelo fue exhibido exitosamente a los clientes y actualmente sólo los miembros de la DECD tienen copias de este modelo.

II.1.2 Plataforma petrolera

Lo que inició como un ejercicio de 6 horas terminó por convertirse en una plataforma petrolera con fines demostrativos. La grúa fue concluida dentro del tiempo sentado para el ejercicio y las modificaciones fueron añadidas en tan sólo otro par de horas.

Se trabajó únicamente en la grúa. El brazo, modelado por un colega, fue adjunto a la cabina, misma que fue modelada a partir de cubos sin textura, color especular rojo y un nivel medio de brillantez como base. El vidrio fue construido a partir de un cubo modificado con escalamiento y con cierto grado de transparencia.

Fig. 2.2: Cabina de la grúa.

El brazo fue adjuntado una vez terminada la cabina. Dicho elemento incluía un script con el cual el usuario, mediante comandos por el chat en un canal selecto, podía girarla en múltiplos de cinco grados.

Se construyó un centro de mando sencillo con una textura tomada de una imagen en una búsqueda por google.com y se añadieron dos elementos con instrucciones para utilizarla. A continuación, se programó un script para facilitar la interacción con los usuarios.

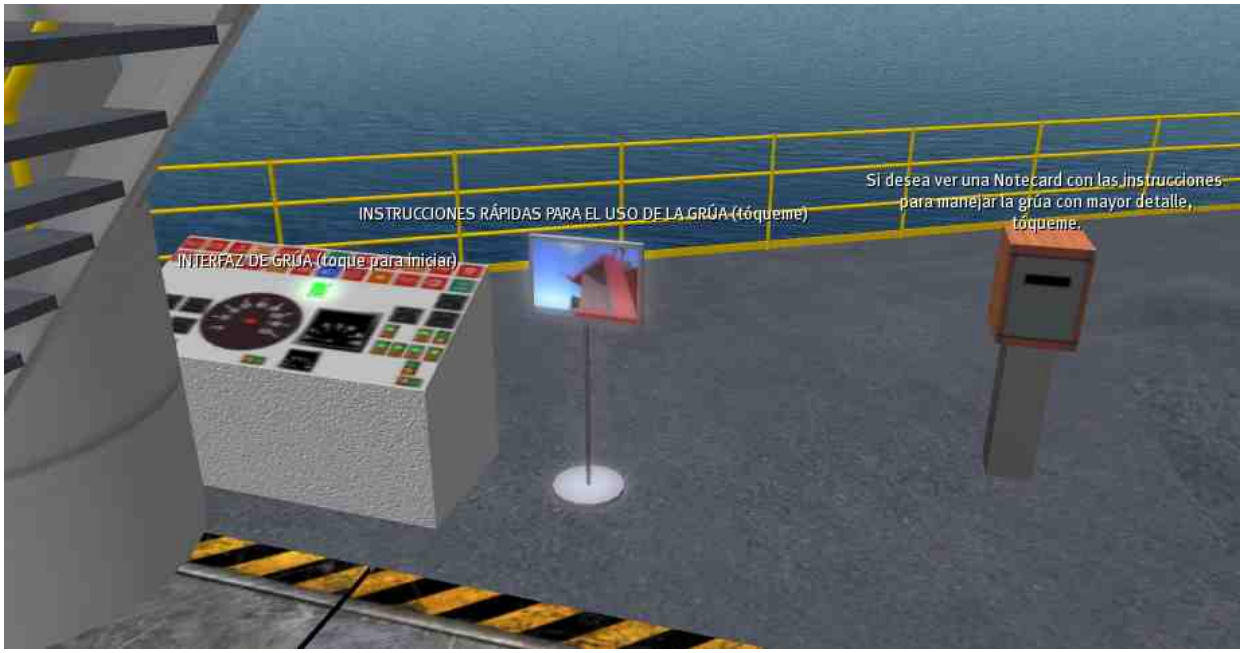


Fig. 2.3: Centro de mando.

A continuación, se presenta el código presente en la grúa:

```
rotation rot;
vector xyz;
vector angrad;
integer escucha;
float Rotacion;
float Rotacion2;
list grados;
integer cont;
float numgrad;

default
{
    state_entry()
    {
        escucha    =    llListen(2000,    "Panel    Control",    NULL_KEY,
"" );//definimos el canal y objeto


        Rotacion = 0;
```

```
    }

    listen(  integer  channel,  string  name,  key  id,  string
message )//escuchamos
    {
        grados = llParseString2List(message,[" "],[]);

        if(llList2String(grados, 0) == "Gira")
        {
            grados  =  llParseString2List(llList2String(grados,  1),["
"],[]);
            numgrad = ((float)llList2String(grados, 0));
            numgrad = numgrad/5;
            for(cont = 0; cont < numgrad; cont++)
            {
                Rotacion = Rotacion + 5; //aumento en los grados
                if(Rotacion == 360)
                {
                    Rotacion = 0;
                }
                xyz = <0,0,5>; // Cuantos grados y en que ejes se rota
                angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
                rot  =  llEuler2Rot(angrad);  //  Calcula  la  matriz  de
rotacion
                llSetLocalRot(llGetRot()*rot); //Rota
                //llSleep(1);
            }
        }

        if(llList2String(grados, 0) == "Arriba")
        {
            grados  =  llParseString2List(llList2String(grados,  1),["
"],[]);
            numgrad = ((float)llList2String(grados, 0));
            numgrad = numgrad/5;
            for(cont = 0; cont < numgrad; cont++)
            {
                if(Rotacion2 > -45)
                {
                    //llSay(0,(string)local);
                    Rotacion2 = Rotacion2 - 5; //aumento en los grados
                    xyz = <0,-5,0>; // Cuantos grados y en que ejes se
rota
                    angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
                    rot = llEuler2Rot(angrad); // Calcula la matriz de
rotacion
                    llSetLocalRot(rot*llGetLocalRot()); //Rota
                    //llSleep(1);
                }
            }
        }

        if(llList2String(grados, 0) == "Abajo")
        {
            grados  =  llParseString2List(llList2String(grados,  1),["
"],[]);
            numgrad = ((float)llList2String(grados, 0));
            numgrad = numgrad/5;
            for(cont = 0; cont < numgrad; cont++)
            {
                if(Rotacion2 < 0)
                    {
```

```
                    //llSay(0,(string)local);
                    Rotacion2 = Rotacion2 + 5; //aumento en los grados
                    xyz = <0,5,0>; // Cuantos grados y en que ejes se
rota
                    angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
                    rot = llEuler2Rot(angrad); // Calcula la matriz de
rotacion
                    llSetLocalRot(rot*llGetLocalRot()); //Rota
                // llSleep(1);
                    }
            }
        }

        if(llList2String(grados, 0) == "Derecha")
        {
            grados   =   llParseString2List(llList2String(grados,   1),["
"],[]);
            numgrad = ((float)llList2String(grados, 0));
            numgrad = numgrad/5;
            for(cont = 0; cont < numgrad; cont++)
            {
                if(Rotacion2 > -45)
                {
                    //llSay(0,(string)local);
                    Rotacion2 = Rotacion2 - 5; //aumento en los grados
                    xyz = <0,0,-5>; // Cuantos grados y en que ejes se
rota
                    angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
                    rot = llEuler2Rot(angrad); // Calcula la matriz de
rotacion
                    llSetLocalRot(rot*llGetLocalRot()); //Rota
                    //llSleep(1);
                }
            }
        }

        if(llList2String(grados, 0) == "Izquierda")
        {
            grados   =   llParseString2List(llList2String(grados,   1),["
"],[]);
            numgrad = ((float)llList2String(grados, 0));
            numgrad = numgrad/5;
            for(cont = 0; cont < numgrad; cont++)
            {
                if(Rotacion2 < 0)
                    {
                    //llSay(0,(string)local);
                    Rotacion2 = Rotacion2 + 5; //aumento en los grados
                    xyz = <0,0,5>; // Cuantos grados y en que ejes se
rota
                    angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
                    rot = llEuler2Rot(angrad); // Calcula la matriz de
rotacion
                    llSetLocalRot(rot*llGetLocalRot()); //Rota
                // llSleep(1);
                    }
            }
        }

        if(message=="reset")
        {
```

```
            xyz = <0,0,-Rotacion>; // Cuantos grados y en que ejes se
rota
            angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
            rot = llEuler2Rot(angrad); // Calcula la matriz de rotacion
            llSetLocalRot(llGetRot()*rot); //Rota
            Rotacion = 0;
        }
    }
}
```

Como podemos ver, la grúa se limita a girar en intervalos de 5 grados, además de que tiene límites de rotación. Para girar hacia arriba, emplea el eje *y* como pivote, mientras que para girar a los lados, toma como eje de referencia al eje *z*.

Y ahora, vemos el código en el objeto "Panel Control", que es el objeto que interactúa directamente con la grúa e interpreta las instrucciones del usuario para comunicárselas a la grúa:

```
key llaveavat;
string name;
integer escucha;
integer iniciook = 0;
integer reloj = 0;

default
{

    state_entry()
    {
        llSetText("INTERFAZ    DE    GRÚA    (toque    para    iniciar)",
<1.0,1.0,1.0>,1.0);
        llSetPrimitiveParams([PRIM_GLOW, ALL_SIDES, 0.2]);
        llSetTimerEvent(0.0);
    }

    touch_start(integer total_number) //El script da inicio al ser tocado
el objeto.
    {
        if (iniciook == 0)
        {
            llaveavat = llDetectedKey(0); //Registra la llave del avatar
que ha tocado el objeto.
            name = llDetectedName(0); //Detecta el nombre del avatar.
            llSay(0,name + ", por favor, introduce la dirección y la
cantidad de grados que deseas mover la grúa (múltiplos de 5 solamente).
\n Utiliza el siguiente formato: Dirección Grados (p.ej.: Arriba 50). \n
Escriba Fin para terminar.") ;
            escucha = llListen(0,"",llaveavat,""); //Entra en estado de
escucha para continuar recibiendo instrucciones del avatar.
            llSetPrimitiveParams([PRIM_GLOW, ALL_SIDES, 0.0]);
            iniciook = 1;
            reloj = 0;
            llSetTimerEvent(1.0);
        }
    }

    listen(integer channel, string name, key id, string message)
    {
        llShout(2000,message); //Si escucha un mensaje, lo retransmite en
el canal 2000, mismo canal que la grúa escucha.
        reloj = 0; //Reinicia el cronómetro.
```

```
        //llListenRemove(escucha);
        if (message == "Fin")
        {
            llSay(0,"¡Hasta pronto!");
            llSay(2000,"Reset");
            llResetScript(); //Si el mensaje es 'Fin', manda un mensaje
de reinicio y se reinicia él mismo.
        }
    }
    timer()
    {
        reloj++;
        if(reloj >= 120)
        {
            llSay(0,"¡Hasta pronto!");
            llSay(2000,"Reset");
            llResetScript(); //Si pasan dos minutos en los que no hay
actividad por parte del usuario, manda un mensaje de reinicio y se
reinicia él mismo.
        }
    }
}
```

Como podemos observar por el código, la función del Panel de Control únicamente consiste en retransmitir los mensajes por el canal que la grúa está escuchando. Esto es una comunidad para el usuario, para evitar que él ingrese manualmente el canal y para que dicho mensaje llegue, con seguridad, al "rango de audición" de la grúa, pues si el avatar diera el mensaje directamente a la grúa, es posible que se encontrara fuera de dicho rango y el mensaje pasara inadvertido.

Esta plataforma ha sido utilizada en numerosas ocasiones para demostrar las capacidades de Second Life en cuanto a propósitos de recorridos virtuales o simulaciones. A la fecha continúa existiendo en una isla privada de la UNAM.

II.1.3. Recolección de hidrocarburos

Este trabajo se programó luego de una visita a las instalaciones de Pemex en Poza Rica. El objetivo era hacer una reconstrucción virtual de dichas instalaciones, que fueran capaces de mostrar mediciones para propósitos de simulacros.

A pesar de la naturaleza simple del proyecto, el vasto modelado geométrico de la instalación consumió mucho tiempo, por lo cual dicho trabajo finalizó – en dicho aspecto (es decir, ignorando programación) – al cabo de dos semanas.

Durante el recorrido, se tomaron varias fotografías de referencia para definir la ubicación y tamaño de las tuberías, bombas y tanques que habían de ser modelados.

La participación en este proyecto fue estrictamente de modelado geométrico; los códigos fueron añadidos posteriormente por los colegas involucrados en dicha área.

Se modelaron una serie de tuberías, las cuales son un conjunto de cilindros combinados con toroides fragmentados. Dado que la isla estaba "desierta" al inicio del proyecto, fue posible

detallar con pernos algunas partes de los modelos, como podemos ver a continuación en la siguiente imagen de una bomba:

Fig. 2.4: Bomba de petróleo.

Este proyecto fue realizado para proveer una sencilla demostración y actualmente ya no es requerido ni utilizado.

II.1.4. Ajedrez

La *Dirección General de Actividades Deportivas y Recreativas (DGADyR UNAM)* propuso un torneo de Ajedrez en línea, usando como plataforma de interacción Second Life. El objetivo radicó en realizar un tablero con piezas en el cual dos participantes pudieran jugar; debía existir un límite de tiempo para ambos jugadores y debía tomarse en cuenta la presencia de un juez para determinar ciertas decisiones de situaciones que bien podrían ser controversiales.

Se otorgó un mes de plazo para concluir este proyecto.

Mientras que tres colegas estuvieron a cargo de programar la funcionalidad en el tablero para poder jugar conforme a las reglas del juego, fue necesario modelar las piezas de ajedrez, – basándose en las piezas *Stauton*, mismas que son utilizadas en los torneos – con sculpties y prims, procurando que tuvieran los menos elementos posibles debido al espacio limitado que existía en la isla en aquel momento y considerando la presencia de al menos diez tableros con todas sus piezas incluidas.

Anteriormente se había recurrido a Blender para el modelado de sculpties. Sin embargo, en esta ocasión se decidió emplear el programa *Rokuro*, mismo que crea una imagen .tga para formar un sólido de revolución.

Fig. 2.5: Tablero con figuras de ajedrez.

La primera pieza modelada, el peón, fue conformado por un único sculptie. El resto de las piezas fueron complementadas con algunos prims, como un par de cubos escaldados y cuyas propiedades fueron modificadas para adornar la parte superior de la torre. El caballo es una pieza compuesta de 5 prims, siendo una de ellas un sculptie. A pesar de que la meta inicial era formar las piezas a partir de un solo sculptie únicamente, se cumplió con el objetivo sentado al no poder consumar el anterior: los peones, siendo la pieza que más se repite a lo largo de los tableros, están conformados únicamente por un sculptie.

Una vez terminados los modelos de las piezas, fue necesario incluir un cronómetro para indicar a ambos jugadores el tiempo restante.

El cronómetro fue modelado a partir de un cubo y fue usada una textura de reloj tomada de la imagen de un cronómetro auténtico. Se adicionó un botón que indicaba, mediante una elevación en su grado de brillantez, si el jugador tenía su turno activo o no.

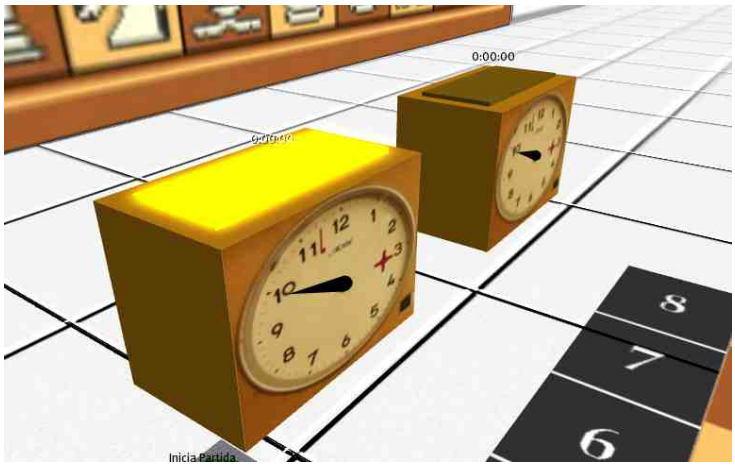


Fig. 2.6: Cronómetros.

El objetivo sentado al programar el cronómetro fue que éste desplegara el tiempo restante del jugador, mismo que podía ser modificado inicialmente por un juez del torneo. Una vez iniciada la partida, el tiempo para el jugador con turno activo disminuía. Cuando un participante terminaba su jugada, su cronómetro se detenía y el cronómetro del contrario comenzaba a funcionar, disminuyendo su tiempo de igual forma.

El formato estaba dispuesto como horas : minutos, siempre y cuando quedara más de una hora de tiempo; en caso contrario, el formato cambiaba a minutos : segundos. Si a un jugador se le agotaba el tiempo, la partida terminaba.

A continuación se presenta el código encontrado en el cronómetro perteneciente al jugador de las piezas blancas:

```
//Declaración de variables.

integer canal;
integer tiempoRestante = -1;
integer horas;
integer minutos;
integer segundos = 59;
integer tiempoOK = 0;
integer turnOn = 1;
integer limite;
integer escucha;
integer inicio = 0;
list speak;

default
{
    state_entry()
    {
        //Estado default. El jugador de blancas inicia con el turno y el
tiempo activos cuando la partida comienza.
        llSetTimerEvent(1.0);
        llSetText("0:00:00",<1.0,1.0,1.0>,1.0);
        //escucha = llListen(canal,"", "","");
    }

    on_rez(integer param)
    {
        canal = param;    //Recibe dentro del parametro del on_rez el
valor del canal de comunicacion a usar
        escucha = llListen(canal-1,"","","");  //Configura el canal de
comunicacion en canal
    }

    timer()
    {

        //Cálculo de las horas.
        horas = tiempoRestante/3600;
```

```
        //Cálculo de los minutos. Si el tiempo restante excede la hora,
se hace el siguiente cálculo para obtener los minutos de la hora
restantes.
        if(horas >= 1)
        {
            minutos = (tiempoRestante%3600)/60;
        }

        //Si el tiempo no excede la hora, se obtienen los minutos de modo
más sencillo.
        if(horas < 1)
        {
            minutos = tiempoRestante/60;
        }

        //Si quedan más de 10 minutos de juego, el cronómetro muestra el
tiempo restante en formato horas - minutos.
        if(tiempoRestante > 600)
        {
            llSetText((string)horas                                    +":"+
(string)minutos,<1.0,1.0,1.0>,1.0);
            segundos--;
            if(segundos < 0)
            {
                segundos = 59;
            }
            tiempoRestante--;
        }

        //Si quedan menos de 10 minutos de juego, el cronómetro muestra
el tiempo restante en formato minutos - segundos.
        if(tiempoRestante <= 600)
        {
            llSetText((string)minutos          +          ":"          +
(string)segundos,<1.0,1.0,1.0>,1.0);

            segundos--;
            if(segundos < 0)
            {
                segundos = 59;
            }
            tiempoRestante--;
        }

        //Si se agota el tiempo del jugador, se manda un mensaje de fin
de juego.
        if(tiempoRestante == -1)
        {
            llSay(2000,"Jaquemate");
        }

        //Cuando el juego no está en curso, se despliega este formato.
        if(tiempoRestante < 0)
        {
            llSetText("0:00:00",<1.0,1.0,1.0>,1.0);
        }
    }

    listen(integer channel, string name, key id, string message)
    {
        speak = llParseString2List(message,[" "],[]);
```

```
        //Cuando el juez ingresa el texto "Tiempo #:#" en el canal
correcto, se asignan las horas y los minutos que va a durar la partida.
Sólo puede hacerse una vez.
        if(llList2String(speak, 0) == "Tiempo")
        {
            if (tiempoOK == 0)
            {
                speak      =      llParseString2List(llList2String(speak,
1),[":"],[]);
                limite   =   ((integer)llList2String(speak,   0)*3600   +
(integer)llList2String(speak, 1)*60);
                tiempoOK = 1;
            }

            else
            {
                llSay(0,"Ya se ha fijado el límite de tiempo");
            }
        }

        //Cuando el juez ingrese el texto "Inicia" en el canal correcto,
comienza la partida. Sólo puede darse el comando una vez.
        else if(message == "Inicia" && inicio == 0)
        {
            tiempoRestante = limite;
            segundos = 0;
            inicio = 1;
        }

        //Cuando un jugador termina de mover su pieza, el tablero debe
mandar este mensaje. Así, inicia el turno del jugador de negras y el
cronómetro se pausa. Si es el turno del jugador de blancas, el cronómetro
continúa.
        else if(message == "Cambia")
        {
            if(turnOn == 1)
            {
                llSetTimerEvent(0.0);
                turnOn = 0;
            }
            else
            {
                llSetTimerEvent(1.0);
                turnOn = 1;
            }
        }

        else if (message == "Mueran!!!")
        {
            llListenRemove(escucha);
            llDie();
        }

        //Si se recibe este mensaje, se rebota en el canal correcto para
asegurar que el otro cronómetro también se detenga y se reinicia el
script (el reenvio del mensaje puede ser prescindible si el tablero lo
envía).
        else if(message == "Jaquemate")
        {
            llSay(2000,"Jaquemate");
            llResetScript();
        }
```

```
    }
}
```

El código para el jugador de las piezas negras es esencialmente el mismo, por lo que no es necesario incluirlo en este reporte.

Para complementar el cronómetro, se añadió el modelo sencillo de una aguja que iba girando en sentido de las manecillas del reloj cuando el tiempo del jugador estaba activo. Éste es el código para la aguja del cronómetro del jugador de las piezas blancas:

```
//Declaración de variables.

integer canal;
integer turnOn = 0;
integer escucha;
integer iniciaOk = 0;
rotation rotG;
vector xyzG;

//Función de rotación local.
rotacionLocal(vector grados)
{
    rotation rot;
    vector xyz;
    vector angrad;
    xyz = grados; // Cuantos grados y en que ejes se rota
    angrad = xyz*DEG_TO_RAD; // Pasamnos a radianes
    rot = llEuler2Rot(angrad); // Calcula la matriz de rotacion
    llSetLocalRot(rot*llGetLocalRot()); //Rota
}

default
{
    state_entry()
    {
        //La aguja del cronómetro de blancas empieza inactiva.
        llSetTimerEvent(0.0);
        //escucha = llListen(2000,"", NULL_KEY,"");
    }

    on_rez(integer param)
    {
        canal = param;    //Recibe dentro del parametro del on_rez el
valor del canal de comunicacion a usar
        escucha = llListen(canal-1,"","",""); //Configura el canal de
comunicacion en canal
    }

    timer()
    {
        //Cuando el cronómetro está activo, rota 6 grados por segundo.
        rotacionLocal(<-6, 0, 0>);
    }

    listen(integer channel, string name, key id, string message)
    {

        //Cuando hay un cambio de turno, la aguja se pausa o se vuelve a
activar.
```

```
        if(message == "Cambia")
        {
            if(turnOn == 1)
            {
                llSetTimerEvent(0.0);
                turnOn = 0;
            }
            else
            {
                llSetTimerEvent(1.0);
                turnOn = 1;
            }
        }

        //Cuando el juez da inicio a la partida, se activa la aguja del
cronómetro. Sólo ocurre una vez.
        if(message == "Inicia")
        {
            if(iniciaOk == 0)
            {
                llSetTimerEvent(1.0);
                iniciaOk = 1;
                turnOn = 1;
            }
            else
            {
            }
        }

        //Cuando el juego termina, la aguja vuelve a su posición original.
        if(message == "Jaquemate")
        {
            xyzG = <0.0, 0.0, 0.0>;
            rotG = llEuler2Rot(xyzG);
            llSetLocalRot(rotG);
            llResetScript();
        }

        else if (message == "Mueran!!!")
        {
            llListenRemove(escucha);
            llDie();
        }
    }
}
```

A pesar de que el proyecto concluyó satisfactoriamente en el tiempo de desarrollo establecido, desafortunadamente jamás fue utilizado para ninguna competencia.

II.1.5. Estación total

Este proyecto es aquel que se desarrolló con el objetivo de crear un objeto que simulara prácticas selectas para un curso de Topología. Se otorgaron dos semanas de plazo para concluir este trabajo.

Lo primero en realizarse fue el modelo geométrico de la estación total, misma que es un conjunto de cubos y cilindros escalados en diferentes ejes y sin texturas; simplemente se añadió efecto de brillo para simular un material metálico.

Un colega desarrolló el modelado geométrico de las dianas que serían usadas en las prácticas.

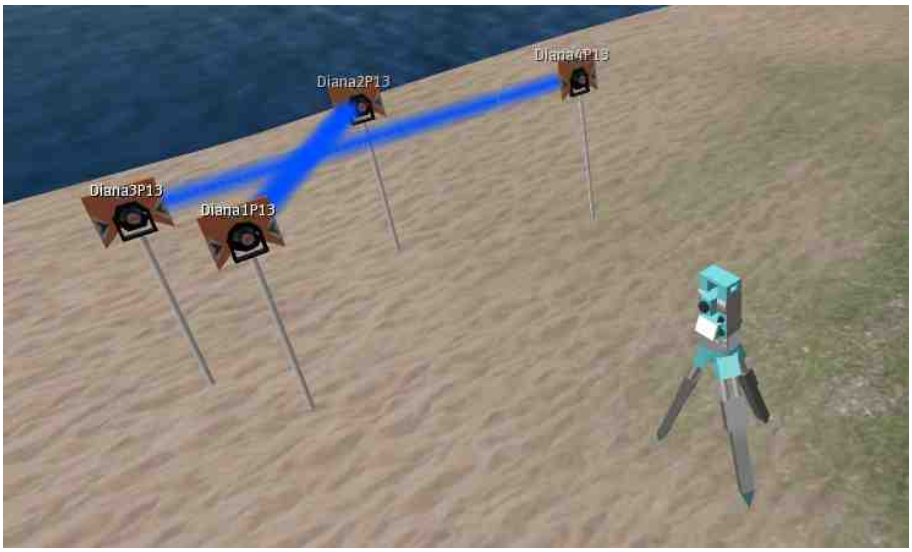


Fig. 2.7: Estación total con dianas.

Las dianas contienen un script que les permite "disparar partículas", ya sea a la estación total o entre ellas, dependiendo de cuál sea la relación que es evaluada (si entre una diana y otra o entre la diana y la estación total).

A continuación, se presenta el código para las dianas en dos scripts separados, desarrollados por un colega:

```
integer canal=751;
integer desicion;
vector Posicion;

default
{
    state_entry()
    {
        llListen(canal,"",NULL_KEY,"");             //Abrimos un canal de
escuha
    }

    listen(   integer   channel,   string   name,   key   id,   string
message )//escuchamos
    {
        if(message == llGetObjectName())
        {
            Posicion = llGetPos();
            desicion = (integer)llFrand(1.99999);
            if(desicion == 0)
```

```
            {
                Posicion.z = Posicion.z - llFrand(0.4999999999);
            }
            else
            {
                Posicion.z = Posicion.z + llFrand(0.4999999999);
            }
            llRegionSay(canal,(string)Posicion);
        }
        if(message == "kill")
        {
            llDie();
        }
    }
}
```

Este script le asigna una altura variable a la diana una vez que su nombre es especificado por un canal no público.

```
integer canal=750;

disparaParticulas(key target)
{

    llParticleSystem([
            PSYS_PART_FLAGS,           (PSYS_PART_EMISSIVE_MASK          |
PSYS_PART_INTERP_COLOR_MASK      |      PSYS_PART_INTERP_SCALE_MASK      |
PSYS_PART_TARGET_LINEAR_MASK  ),
                PSYS_SRC_PATTERN, PSYS_SRC_PATTERN_ANGLE,
                PSYS_SRC_TARGET_KEY, target,
                PSYS_PART_START_COLOR, <0.0,0.3,1.0>,
                PSYS_PART_END_COLOR, <0,0.3,1.0>,
                PSYS_SRC_ANGLE_BEGIN, 0.0,
                PSYS_SRC_ANGLE_END, 0.0,
                PSYS_PART_START_ALPHA, 0.80,
                PSYS_PART_END_ALPHA, 0.80,
                PSYS_PART_START_SCALE, <0.3,0.3,0.3>,
                PSYS_PART_END_SCALE, <0.3,0.3,0.3>,
                PSYS_SRC_BURST_PART_COUNT, 1,
                PSYS_SRC_BURST_RATE, 0.1
        ]);
}

default
{
    state_entry()
    {
        llListen(canal,"",NULL_KEY,"");
    }

    listen(integer channel, string name, key id, string message)
    {
        list mensaje;
        mensaje=llParseString2List(message,["#"],[]);
        if (llList2String(mensaje,0)==llGetObjectName())
        {
            disparaParticulas((key)llList2String(mensaje,1));
        }
        if (message=="off")
        {
```

```
            llParticleSystem([]);
        }
    }
}
```

Este otro inicializa una instancia de partículas, que la diana disparará hacia el objeto que se le indique.

Ahora, se presentarán algunos fragmentos del script perteneciente a la estación total. Debido a la extensión del script, solamente se seleccionarán cuatro prácticas de las trece programadas.

El fragmento principal (no presentado aquí) pone a la estación total en un estado de escucha. El usuario debe escribir el título de una práctica en la barra del chat (practica 1, practica 2, etc.) y al ingresar esto, la estación total cambiará de estado, invariablemente provocando que el objeto despliegue las dianas necesarias para llevar a cabo la práctica seleccionada.

Dicho fragmento de código también presenta algunas funciones que se utilizarán en los cálculos de las prácticas, como punto medio, distancia entre rectas, etc.

Una vez que las dianas estén disponibles, será necesario moverlas manualmente, es decir, seleccionándolas y eligiendo modificar el objeto para desplazarlo hacia donde se desee.

Éste es el fragmento de programación perteneciente a la práctica 1:

```
state p1
{

    state_entry()
    {
        posteodolito = llGetPos();
        escucha = llListen(canal, "", NULL_KEY, "");
        if (rezFlag==1)
        {

llRezObject("Diana1P1",llGetPos()+<1.0,0.0,0.0>,ZERO_VECTOR,ZERO_ROTATION
,0);
            llSleep(0.5);
        }
        llRegionSay(canal,"Diana1P1");

    }

    listen( integer channel, string name, key id, string message )
    {
        posdiana1 = (vector)message;
        llRegionSay(canalParticulas,"Diana1P1#"+(string)llGetKey());
        resultadohor = distanciaHorizontal(posteodolito, posdiana1);
        resultadodesn = desnivel(posteodolito, posdiana1);
         if(message == "inicia")
         {
        llSay(0, "La distancia horizontal es: "+(string)resultadohor+" y
el desnivel es: "+(string)resultadodesn);
        llResetScript();
         }
    }
```

```
}
```

Cuando el usuario escribe la palabra inicia en el chat, el script lleva a cabo la función de calcular la distancia horizontal, dada por la fórmula:

$$d = \sqrt{(Vx2 - Vx1)^2 + (Vy2 - Vy1)^2}$$

Esto dará como resultado la distancia horizontal entre el teodolito y la diana. Esto se visualiza en Second Life como se aprecia en la Fig. 2.8.

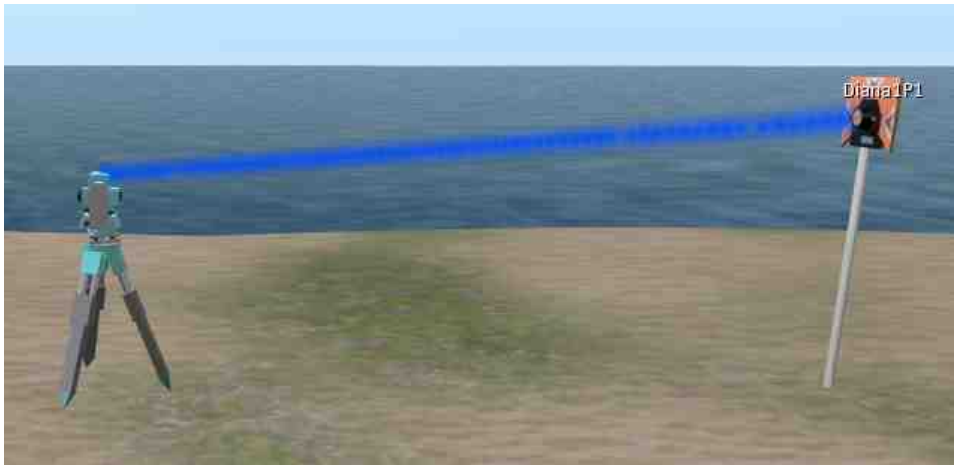


Fig. 2.8: Práctica 1.

Esto concluye la práctica uno. Ahora, se mostrará el fragmento perteneciente a la práctica 2.

```
state p2
{

    state_entry()
    {
        //posteodolito = llGetPos();
        escucha = llListen(canal, "", NULL_KEY, "");
        if (rezFlag==1)
        {

llRezObject("Diana1P2",llGetPos()+<1.0,0.0,0.0>,ZERO_VECTOR,ZERO_ROTATION
,0);

llRezObject("Diana2P2",llGetPos()+<1.0,0.5,0.0>,ZERO_VECTOR,ZERO_ROTATION
,0);
            llSleep(0.5);
        }
        llRegionSay(canal,"Diana1P2");
        //llRegionSay(canal,"Diana2P2");
    }

    listen( integer channel, string name, key id, string message )
    {
           if(name == "Diana1P2")
```

```
            {
                posdiana1 = (vector)message;
                llSay(0,(string)message);
                llRegionSay(canal,"Diana2P2");
            }
            if(name == "Diana2P2")
            {
             posdiana2 = (vector)message;
             llSay(0,(string)message);
             llRegionSay(canalParticulas,"Diana1P2#"+(string)id);
             resultadohor = distanciaHorizontal(posdiana1, posdiana2);
             resultadodesn = desnivel(posdiana1, posdiana2);
              if(message == "inicia")
              {
             llSay(0,       "La       distancia       horizontal       es:
"+(string)resultadohor+" y el desnivel es: "+(string)resultadodesn);
             llResetScript();
              }
            }
    }
}
```

Como podemos observar, la práctica funciona exactamente de la misma manera que la primera. Sin embargo, en vez de hacer la medición de la distancia horizontal con respecto a la estación total, se realiza dicha operación con respecto a una segunda diana.

A continuación, observamos la visualización en Second Life.

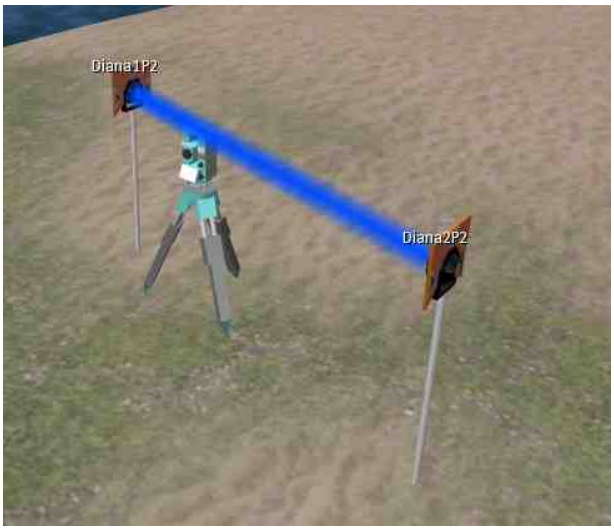


Fig. 2.9: Práctica 2.

Veamos ahora la práctica 7:

```
state p7
{

    state_entry()
    {
```

```
        posteodolito = llGetPos();
        escucha = llListen(canal, "", NULL_KEY, "");
        if (rezFlag==1)
        {

llRezObject("Diana1P7",llGetPos()+<1.0,0.0,0.0>,ZERO_VECTOR,ZERO_ROTATION
,0);
            llSleep(0.5);
        }
        llRegionSay(canal,"Diana1P7");
    }

    listen( integer channel, string name, key id, string message )
    {
        posdiana1 = (vector)message;
        llRegionSay(canalParticulas,"Diana1P7#"+(string)llGetKey());
        resultadohor = distanciaHorizontal(posteodolito, posdiana1);
        resultdistinc = distanciaInclinada(posteodolito, posdiana1);
        resultadodesn = desnivel(posteodolito, posdiana1);
         if(message == "inicia")
         {
        llSay(0, "La distancia horizontal es: "+(string)resultadohor+",
el desnivel es: "+(string)resultadodesn+" y la distancia inclinada es:
"+(string)resultdistinc);
        llResetScript();
         }
    }
}
```

Esta práctica funciona de modo similar a las anteriores, pero a diferencia de éstas, la práctica 7 sí incorpora el eje Z en sus cálculos, ya que no sólo se muestra la distancia horizontal entre la diana y la estación total, sino la distancia inclinada, dada por esta fórmula:

$$d = \sqrt{(Vx2 - Vx1)^2 + (Vy2 - Vy1)^2 + (Vz2 - Vz1)^2}$$

Se calcula de igual forma el desnivel entre la diana y la estación total, la cual está dada por el resultado absoluto de la diferencia de sus posiciones en el eje Z.

La visualización en Second Life nuevamente puede apreciarse en la Fig. 2.10.

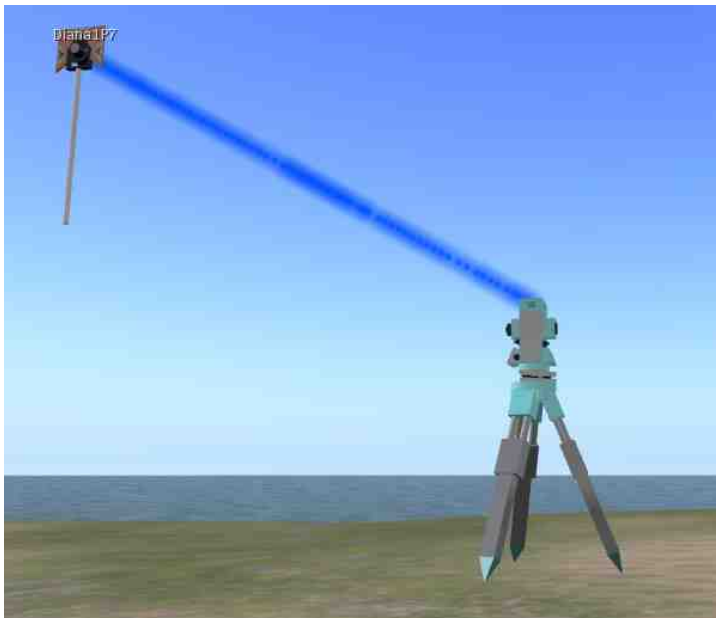


Fig. 2.10: Práctica 7.

Por último, se presenta la práctica 13.

```
state p13
{

    state_entry()
    {
        escucha = llListen(canal, "", NULL_KEY, "");
        if (rezFlag==1)
        {

llRezObject("Diana1P13",llGetPos()+<1.0,0.0,0.0>,ZERO_VECTOR,ZERO_ROTATIO
N,0);

llRezObject("Diana2P13",llGetPos()+<1.0,0.5,0.0>,ZERO_VECTOR,ZERO_ROTATIO
N,0);

llRezObject("Diana3P13",llGetPos()+<1.0,1.0,0.0>,ZERO_VECTOR,ZERO_ROTATIO
N,0);

llRezObject("Diana4P13",llGetPos()+<1.0,0.5,0.0>,ZERO_VECTOR,ZERO_ROTATIO
N,0);
            llSleep(0.5);
        }
        llRegionSay(canal,"Diana1P13");
    }

    listen( integer channel, string name, key id, string message )
    {
           if(name == "Diana1P13")
           {
               posdiana1 = (vector)message;
               llSay(0,(string)message);
               llRegionSay(canal,"Diana2P13");
```

```
            }
            if(name == "Diana2P13")
            {
                 posdiana2 = (vector)message;
                 llSay(0,(string)message);
                 llRegionSay(canalParticulas,"Diana1P13#"+(string)id);
                 llRegionSay(canal,"Diana3P13");

            }
            if(name == "Diana3P13")
            {
                 posdiana3 = (vector)message;
                 llSay(0,(string)message);
                 llRegionSay(canal,"Diana4P13");
            }
            if(name == "Diana4P13")
            {
                posdiana4 = (vector)message;
                llSay(0,(string)message);
                llRegionSay(canalParticulas,"Diana3P13#"+(string)id);
                paralel = paralelas (posdiana1, posdiana2, posdiana3,
posdiana4);
         if(message == "inicia")
         {
                if(paralel == 1)
                {
                    llSay(0,"No hay punto de intersección: las rectas son
paralelas.");
                    llResetScript();
                }
                else
                {
                    pospunto = interseccion (posdiana1, posdiana2,
posdiana3, posdiana4);
                    llSay(0,"El punto de intersección es X =
"+(string)pospunto.x+" y Y = "+(string)pospunto.y);
                    llResetScript();
               }
                }
            }
    }
}
```

Esta práctica identifica el punto de intersección creado entre las rectas formadas por dos pares de dianas.

Ante todo, se revisa si dichas rectas no son paralelas a partir de una operación de producto cruzado de vectores. De ser el caso, el programa informará al usuario de dicha condición. Si existe un punto de intersección, entonces éste es calculado mediante la obtención de la fórmula de la recta:

$$y = mx + b$$

Esto se logra igualando los puntos, obtenidas la pendiente y el punto de corte de la recta con el eje Y.

Se presenta la visualización en la Fig. 2.11.

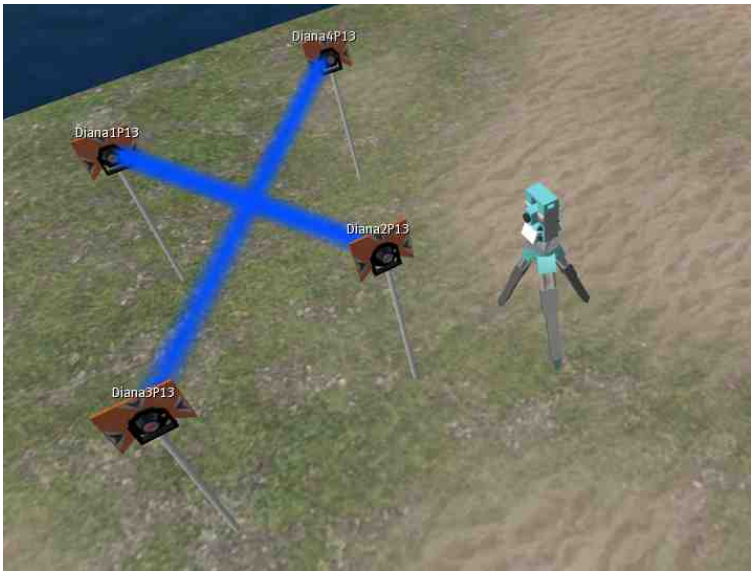


Fig. 2.11: Práctica 13.

La estación total fue terminada dentro del tiempo concedido y fue empleada exitosamente durante un curso demostrativo.

II.1.6 Juego de la cerveza

El último de los proyectos a discutir es el llamado juego de la cerveza, del cual se realizó una simulación meramente demostrativa dentro de Second Life, con el fin de mostrar a los interesados la dinámica del juego. Este trabajo fue desarrollado a lo largo de una semana.

Aunque laborioso en cuanto a modelado geométrico y programación, el proyecto del juego de la cerveza es relativamente sencillo, pues el código que emplea no contiene sino una variable y consta de eventos completamente predeterminados.

Ante todo, es imperativo explicar en qué consiste el juego de la cerveza, pues sus reglas se aplicaron a la demostración:

> Se inicia el juego con 5 jugadores: la fábrica, que para fines demostrativos, goza de producción infinita; la agencia; la sub-agencia; el detallista y, finalmente, el consumidor.
>
> Los cuatro primeros reciben una cantidad específica de “cervezas”.
>
> El moderador, quien tomará el papel del consumidor, realizará un pedido al detallista de la cantidad de unidades que él designe. El detallista está obligado, si

tiene las unidades necesarias para satisfacer la demanda, a entregar el pedido o al menos parte de él. Entonces, el mismo detallista tiene el derecho de realizar un pedido a la sub-agencia – la entidad inmediatamente superior – por las unidades que desee. Sin embargo, a diferencia del pedido del consumidor al detallista, el cual es inmediato, el pedido del último a la sub-agencia tiene un retraso de dos semanas, es decir, dos turnos.

Al cabo de estos dos turnos, durante los cuales el consumidor continuará realizando pedidos al detallista, la sub-agencia podrá realizar un envío de las unidades requeridas, no más, no menos. Dicho pedido tendrá de igual forma un retraso de dos semanas en llegar a su destino. A su vez, la sub-agencia seguirá la misma dinámica al hacer un pedido del número de unidades que suponga necesaria a la agencia. Esta solicitud también tardará dos turnos en llegar a la siguiente entidad.

Esta misma dinámica será seguida por la agencia, quién realizará el pedido a la fábrica que, como ya se ha establecido, posee capacidad de producción infinita y representa el fin de la cadena de pedidos.

Al finalizar un turno, cada entidad debe presentar una tabla indicando cuántas fueron las unidades que entregaron, cuántas fueron las unidades solicitadas, cuál fue el número de unidades que no pudieron entregarse y, finalmente, cuántas unidades continúan en la posesión de la entidad (para propósitos demostrativos, estas estadísticas serán visibles para todos, pero en realidad, únicamente el moderador podrá verlas).

Se utilizaron modelos previamente realizados para la construcción de una maqueta demostrativa del ciclo del agua. La gran mayoría constituyen simples cubos con texturas sencillas, mapeadas de forma muy básica. Sólo fue necesaria la distribución de éstos a través de las 16 cuadras que formaron la maqueta.

Fig. 2.12: Maqueta demostrativa.

Sin embargo, sí fue necesario modelar algunos edificios, como lo son la fábrica y el centro detallista. También se realizaron modelos de dos tipos de camiones distribuidores. Todos los modelos fueron hechos con cubos modificados y cilindros en el caso de los neumáticos de los camiones. Adicionalmente, se crearon texturas utilizando la marca de cerveza Modelo (ya que serían ellos quienes recurrirían a esta demostración) y de la tienda de abarrotes Extra (debido a que es distribuidora oficial de la cerveza mencionada).

Fig. 2.13 y 2.14: Fábrica con camiones y detallista.

Se añadió una "valla" alrededor de la ciudad para bloquear parte del Palacio de Minería virtual a los cursantes y un par de botones: uno iniciaba una animación, que estaba compuesta de efectos de partículas que indicaban una dirección. Estas direcciones representan el flujo de los pedidos de cerveza de la planta de producción infinita (la fábrica) hasta el detallista (la tienda de abarrotes Extra), apegándose a la dinámica del juego de la cerveza. Se utilizaron diferentes colores para este efecto: partículas de color azul para representar el viaje del camión de entrega desde la fábrica hasta la agencia; verde para representar el viaje desde ésta a la sub-agencia; rojo para indicar la entrega a las tiendas detallistas y, finalmente, naranja para representar la compra del consumidor al detallista (para este efecto, se utilizaron los cubos desarrollados para el

proyecto de la *Estación total* para las prácticas de topología). Una vez iniciada esta animación, podía ser detenida presionando el segundo botón.

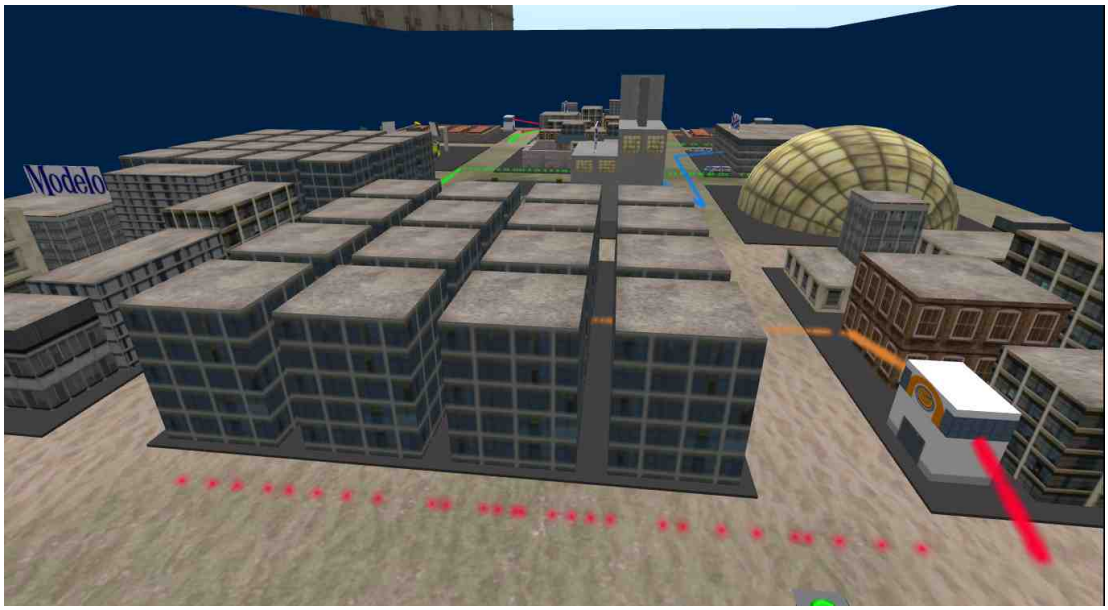


Fig. 2.15: Maqueta con efectos de partículas encendidos.

Se hicieron modelos muy básicos de "jugadores" a partir de cilindros. Sólo se usaron colores sin texturas y una brillantez media para dar efecto de maniquíes metálicos. Cada modelo posee un torso de color diferente: el jugador de la fábrica "viste" azul; la agencia verde; la sub-agencia rojo; el detallista naranja y, por último, el consumidor muestra un color blanco. Estos cinco modelos, idénticos entre sí salvo por el detalle mencionado, se encuentran sentados alrededor de una mesa redonda cerca unos de otros, la cual está provista de un asiento donde el usuario que iniciará la demostración puede sentarse. Una vez que el avatar tome asiento, la cámara se fijará en un ángulo que permita enfocar a todos los jugadores.

Existen varios trozos de papel colocados entre jugadores, así como una caja asignada a cada uno de ellos. Dentro de cada caja, existen modelos de latas de cerveza – creados a partir de simples cilindros con una textura de la marca Modelo – que irán moviéndose entre jugadores de acuerdo a las reglas sentadas por el juego.

Los trozos de papel indican un pedido realizado por un jugador a su entidad superior; el mensaje viaja de papel en papel dependiendo del turno actual. Lo mismo ocurre con las latas, con la diferencia de que éstas se mueven gradualmente hacia su destino.

En la mesa, existe un bloque con textura de calendario que indica con texto cuál es la "semana" o "turno" actual. Hay otros dos botones, con forma triangular, que sirven para avanzar o retroceder un turno, con tal de tener la opción de regresar a un estado anterior para propósitos demostrativos. Adicionalmente, existe un botón rectangular de color rojo el cual, de ser presionado, mandará un mensaje que hará que todo el juego regrese a su estado original.

El juego funciona a base de scripts que envían señales para iniciar secuencias de mandos (los botones). El resto de los objetos escucha y, dependiendo del turno, muestran un comportamiento diferente. Los jugadores, al final de cada turno, mostrarán una pequeña tabla que indica cuántas cajas entregaron, cuántas fueron pedidas, cuántas no pudieron ser entregadas y cuántas se deben en total.

Debido a que el juego es enteramente demostrativo y con un nivel de interactividad que no excede la presión de botones, los modelos tienen un comportamiento completamente predecible. No existen variables en las estadísticas, pues todo está conformado por texto previamente establecido.

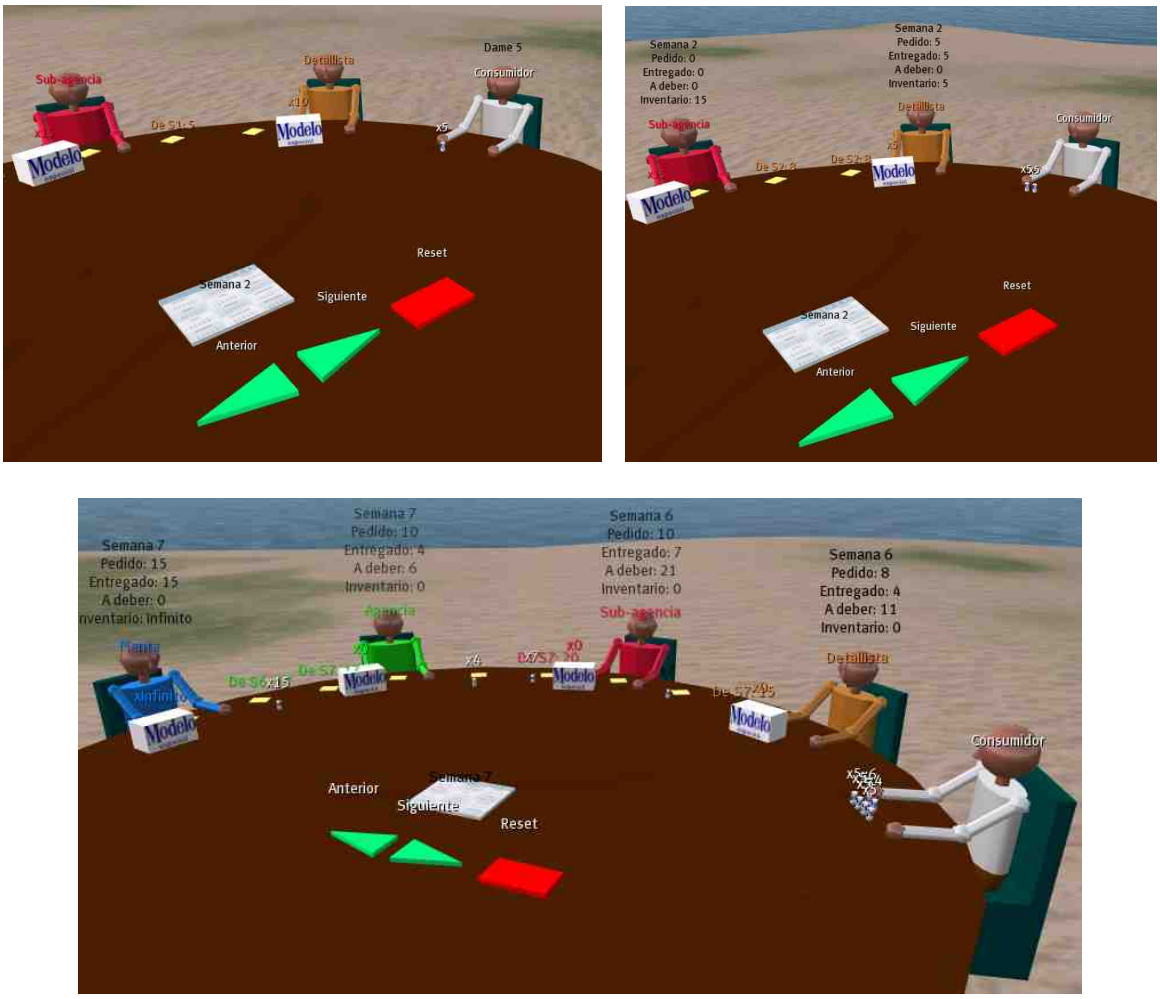


Fig. 2.16 – 2.18: Mesa demostrativa con jugadores.

A continuación, se mostrarán algunos scripts empleados para esta demostración. Primero, se muestra el código del jugador "detallista":

```
integer escucha;
float residuo;
```

```
integer numensaje = 0;

default
{
    state_entry()
    {
        escucha = llListen(200, "", NULL_KEY, "");
    }

    listen( integer channel, string name, key id, string message )
    {
        if(message == "siguiente")
        {
            numensaje++;
            residuo = numensaje%2;
            state habla;
        }
        if(message == "anterior" && numensaje > 0)
        {
            numensaje--;
            residuo = numensaje%2;
            state habla;
        }
        if(message == "Reset")
        {
            llSetText("",<0.87,0.5,0.2>,1.0);
            llResetScript();
        }
    }

}

state habla
{
    state_entry()
    {

        if(residuo != 0)
        {
            if(numensaje >= 15)
            {
                llSetText("",<0.87,0.5,0.2>,1.0);
                llResetScript();
            }
            else
            {
                llSetText("",<1.0,1.0,1.0>,1.0);
                state default;
            }
        }
        else
        {
        if(numensaje == 0)
        {
            llSetText("",<0.87,0.5,0.2>,1.0);
            state default;
        }

            if(numensaje == 2)
            {
                llSetText("Semana  1\nPedido:  5\nEntregado:  5\nA  deber:
0\nInventario: 10",<0.0,0.0,0.0>,1.0);
```

```
                state default;
            }

            if(numensaje == 4)
            {
                llSetText("Semana  2\nPedido:  5\nEntregado:  5\nA  deber:
0\nInventario: 5",<0.0,0.0,0.0>,1.0);
                state default;
            }

            if(numensaje == 6)
            {
                llSetText("Semana  3\nPedido:  6\nEntregado:  5\nA  deber:
1\nInventario: 0",<0.0,0.0,0.0>,1.0);
                state default;
            }

            if(numensaje == 8)
            {
                llSetText("Semana  4\nPedido:  5\nEntregado:  0\nA  deber:
6\nInventario: 0",<0.0,0.0,0.0>,1.0);
                state default;
            }

            if(numensaje == 10)
            {
                llSetText("Semana  5\nPedido:  6\nEntregado:  5\nA  deber:
7\nInventario: 0",<0.0,0.0,0.0>,1.0);
                state default;
            }

            if(numensaje == 12)
            {
                llSetText("Semana  6\nPedido:  6\nEntregado:  6\nA  deber:
7\nInventario: 2",<0.0,0.0,0.0>,1.0);
                state default;
            }

            if(numensaje == 14)
            {
                llSetText("Semana  6\nPedido:  8\nEntregado:  4\nA  deber:
11\nInventario: 0",<0.0,0.0,0.0>,1.0);
                state default;
            }

            else
            {
                llSetText("",<0.87,0.5,0.2>,1.0);
                llResetScript();
            }

        }
    }

}
```

Como podemos ver, los estados son muy sencillos. El “detallista” sólo muestra sus estadísticas entre turnos, mismas que están compuestas por números constantes. Los scripts del resto de los jugadores son prácticamente idénticos al mostrado.

En cuanto a los botones, éstos únicamente emiten mensajes que provocan que el resto de los objetos actúen de acuerdo al contador que cada uno tiene integrado.

Las notas se comportan de igual manera que los jugadores, desplegando su información en diversas fases del turno o mostrándola constantemente, dependiendo de la posición del mensaje.

Por último, la lata funciona de modo similar a los jugadores y los mensajes, pero también se mueve poco a poco, dependiendo del turno y su fase. Cada lata despliega una cantidad diferente.

Se decidió no incluir los scripts de estos objetos debido a su similitud con el ya mostrado.

Esta demostración fue utilizada por el cliente un par de veces para explicar la dinámica del juego y actualmente no es utilizada, debido a que se optó por realizar el juego en Unity.

II.2 UNITY

II.2.1 Laboratorio CDM

Curiosamente, este fue el último proyecto de Unity en dar inicio. No obstante, fue necesario completarlo en dos semanas, lo cual demandó una gran labor, desde la toma de fotografías hasta la creación de imágenes y el modelado geométrico de ciertos objetos para añadir contenido al entorno virtual.

El objetivo del proyecto determinó que se creara una estancia virtual cuyo aspecto estuviera basado en el laboratorio CDM (Centro de Diseño Mecánico) del Anexo de Ingeniería de Ciudad Universitaria, y adicionalmente, que fuera lo suficientemente grande como para poder contener veinte avatares quienes tomarían un curso a distancia en su interior.

El primer paso para lograr la reproducción de este laboratorio fue la toma de fotografías. Se utilizó una cámara de alta resolución, propiedad de la UNAM, para entrar al laboratorio y tomar fotografías y medidas del sitio con ayuda de un metro.

Una vez capturadas las imágenes de referencia esenciales, y anotada la disposición de las mesas de laboratorio y los aparatos, se comenzó a realizar la virtualización del mismo.

Mientras que un grupo de compañeros de trabajo se enfocaron en la realización del modelo del laboratorio del CDM, la virtualización de algunos aparatos y el desarrollo del contenido para el curso en línea, se llevó a cabo la creación de texturas en el software Photoshop.

Tomando el color base de la pared fotografiada, se creó una textura de pared utilizando diversos filtros – añadiendo ruido y dando un efecto de bordes – para crear el efecto de rugosidad. En una capa diferente, se añadió una franja roja para completar el efecto.

Fig. 2.19: Pared del laboratorio.

La creación del suelo no fue diferente: se sentó una imagen base utilizando filtros de ruido y efectos de bordes.

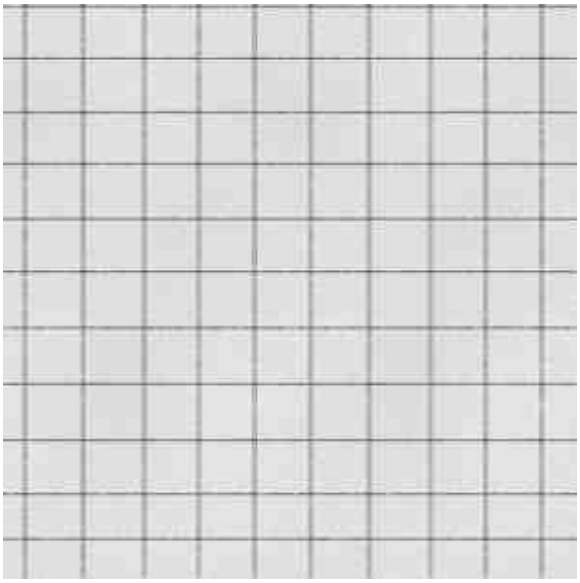


Fig. 2.20: Suelo del laboratorio.

Para las ventanas se siguió el mismo proceso, pero una vez terminada la base, se procedió a dar un efecto de transparencia con una muy ligera pizca de opacidad (misma que simulaba la presencia de un vidrio) y, haciendo uso de la herramienta de bordes, se simularon las partes metálicas.

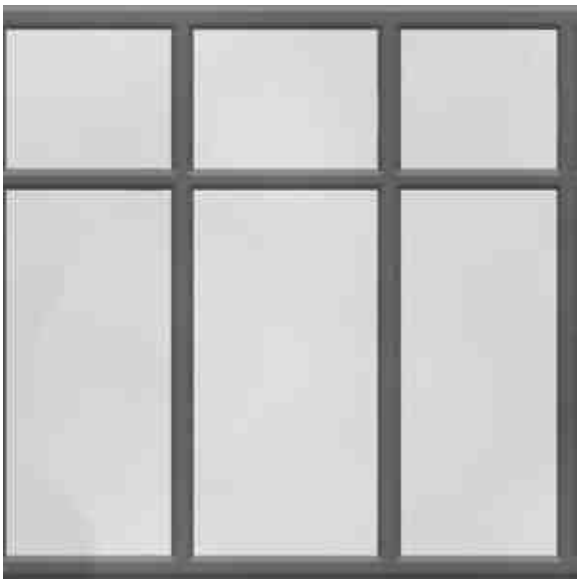


Fig. 2.21: Ventana del laboratorio.

Puesto que el techo tiene la misma distribución que las ventanas, se utilizó el procedimiento mencionado anteriormente.

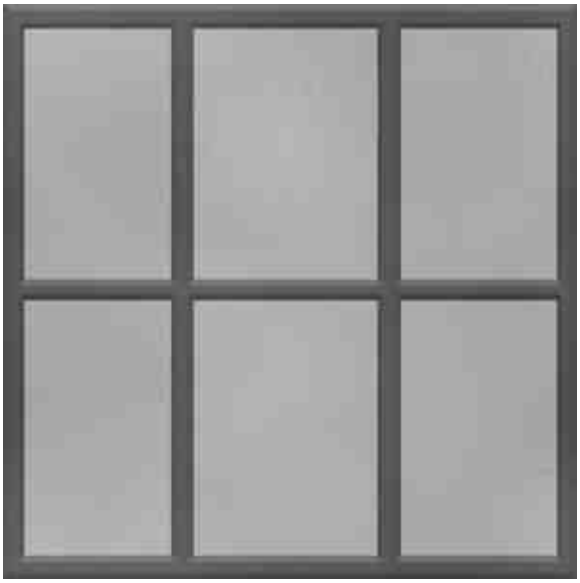


Fig. 2.22: Techo del laboratorio.

La entrada al laboratorio contiene algunos toques adicionales con efecto de bordes para el cerrojo.

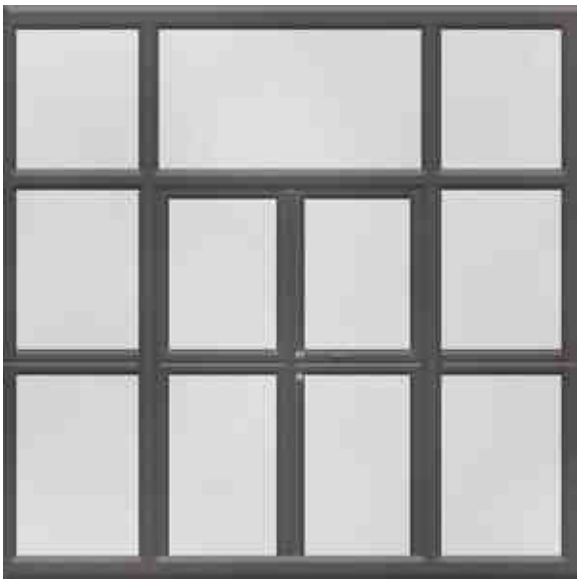


Fig. 2.23: Entrada al laboratorio.

También se crearon texturas para la mesa: una para la superficie y otra para el borde. Ambas fueron procesadas en el software Paint Shop Pro para lograr el efecto de mosaico.

Fig. 2.24: Mesa del laboratorio.

Una vez terminadas las texturas, se procedió a sacar los mapas de normales de la imagen de pared, suelo y aquellas pertenecientes a la mesa; sin embargo, fue necesario crear una imagen en blanco y negro para las texturas que contenían "trozos de metal", ya que el color y efecto de borde de dichas imágenes daba un mapa de normales erróneo. Al construir un mapa de blanco y negro, los bordes se perdían y el software interpretaba la imagen de manera óptima.

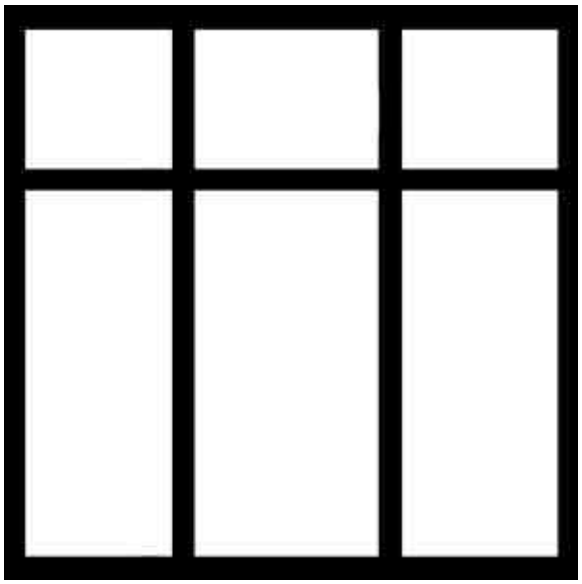
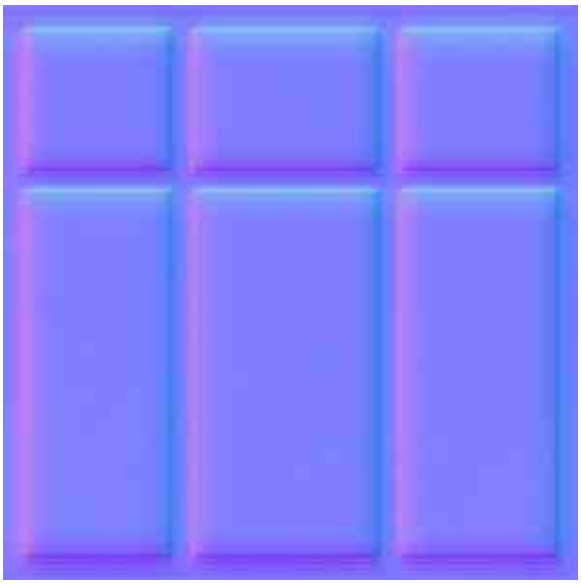



Fig. 2.25 y 2.26: Ventana del laboratorio en blanco y negro y su mapa de normales.

Se realizó la textura de un instrumento de medición del laboratorio. Se asignó un efecto de metal genérico amarillo a la mayor parte de sus piezas, pero fue necesario recrear un panel de indicaciones. Un colega se encargó de hacer baking a la textura una vez integrada al modelo para un mejor efecto.

Fig. 2.27 y 2.28: Textura del instrumento de laboratorio y modelo geométrico de dicho instrumento con baking.

Para complementar la virtualización del laboratorio, se incluyeron aparatos de formas sencillas, tales como una impresora tridimensional, una pizarra y un banco. Estos elementos fueron modelados en 3ds Max a partir de cubos y cilindros, alterados con extrusiones y subdivisiones de sus caras. Se realizó un mapa de UV de cada uno de estos objetos una vez finalizado el proceso de modelado geométrico, mismo que serviría de referencia para la creación de texturas (realizadas en Photoshop); y ya terminadas éstas, se procedió a realizar sus mapas de normales en el software PixPlant.

Fig. 2.29 y 2.30: Impresora tridimensional y banco.

Ahora se presentan imágenes del laboratorio CDM finalizado y funcionando en la plataforma Unity:

Fig. 2.31 – 2.33: Laboratorio CDM completo.

Este proyecto fue utilizado para realizar unos cuantos cursos a larga distancia, por lo cual el objetivo inicial se cumplió satisfactoriamente. A la fecha ya no es utilizado.

II.2.2 Juego de Coca-Cola

A diferencia del proyecto del juego de la cerveza para Second Life, el cuál fue meramente demostrativo, el objetivo de éste era ser interactivo, de modo tal que jugadores humanos pudieran sentarse, cada uno desde una computadora distinta y jugarlo simultáneamente, siendo moderados por el consumidor, quien sería el único capaz de ver las estadísticas individuales que los participantes presentarían automáticamente al final de cada turno. En otras palabras, la dinámica sería exactamente que se siguió en el proyecto anteriormente mencionado. Se otorgaron tres meses para la construcción del ambiente y la programación del mismo.

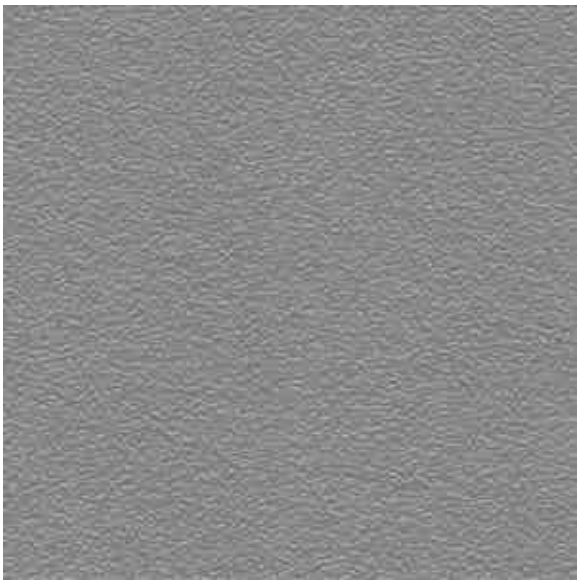
No obstante, solamente se tuvo participación en el área de modelado geométrico, ya que la programación corrió por parte de otro equipo.

Se asignaron tres áreas distintas a cada uno de los integrantes del equipo del modelado geométrico: el primer modelador estuvo encargado de llevar a cabo la virtualización de una fábrica de Coca-Cola. El segundo realizó el modelo geométrico de un área suburbana, un parque, semáforos y el detallista. Finalmente, el presente se encargó de la virtualización de una ciudad, así como el modelado geométrico de la agencia y la sub-agencia.

Se optó por usar modelos muy sencillos con una cantidad baja de polígonos, con tal de dedicarle mayor énfasis a la creación de texturas de alta resolución que, junto con sus mapas de normales, complementarían la simpleza de los modelos.

La vasta mayoría de los edificios son en realidad cubos, mismos que fueron subdivididos y separados en distintos objetos, con tal de lograr un mejor efecto para su textura individual. Cada edificio tiene su propia pared genérica, que es una textura base, y fue utilizada en la construcción de las ventanas y puertas. Se construyeron imágenes en blanco y negro de las imágenes a partir

de las cuales se generarían los mapas de normales, siempre cuidando que el efecto de rugosidad en la pared – de haber uno – fuera visible y generado en dicho mapa.



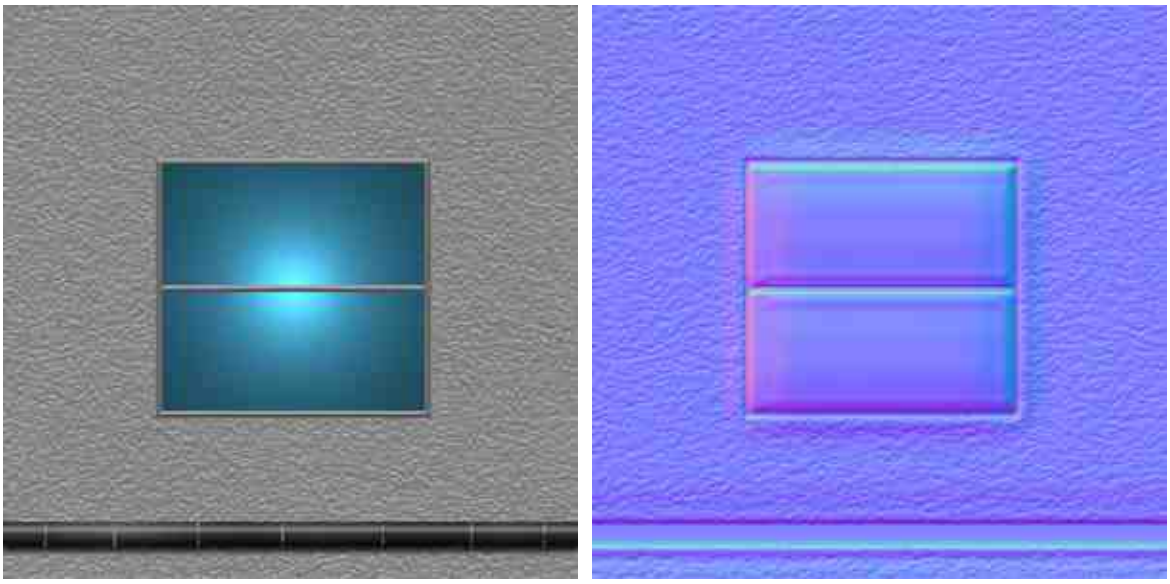


Fig. 2.34 – 2.38: Ejemplos de pared genérica y ventana y puerta, éstas dos junto con sus mapas de normales.

Para acelerar el proceso, se decidió construir solamente los suficientes modelos de edificios como para colocar varios dentro de la ciudad, pero teniendo la precaución de que el efecto no cayera en una flagrante monotonía.

Se tuvieron diez cuadras a cargo. Cuatro de ellas – el "centro" de la ciudad – están compuestas por seis modelos de edificios, cada uno con texturas diferentes. Estos son los edificios más "altos" del modelo.

Fig. 2.39 y 2.40: Toma del centro de la ciudad.

Las cuatro cuadras del noreste de la ciudad simulan un "área residencial", donde los edificios no poseen la misma altura que aquellos encontrados en el centro. Esta zona está provista con un estacionamiento y la sub-agencia. A diferencia de los demás edificios, ésta es un único objeto, con una textura construida tomando como base el mapeado UV del mismo. Se tomó la decisión de no dividir al objeto en varios por su tamaño pequeño. A su lado, podemos encontrar un pequeño "almacén", cuya textura de puerta fue rehusada en la construcción de la agencia.

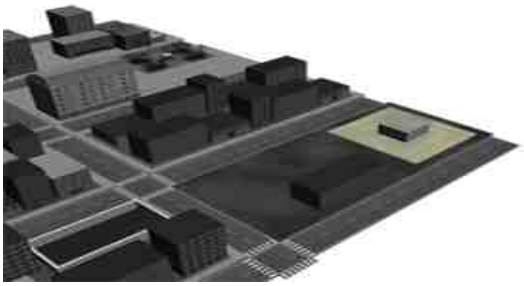


Fig. 2.41 – 2.43: Toma de área residencial y de la sub-agencia.

Se desarrolló también una textura sencilla para las cuadras.

Las dos cuadras restantes están compuestas por la agencia y el estacionamiento de la misma. Debido a la falta de imágenes de referencia, se decidió tomar la agencia como una bodega y se desarrolló un modelo muy sencillo. Éste está compuesto por un simple cubo y la mitad de un cilindro. El cubo está provisto de la misma textura que se utilizó para el almacén de la sub-agencia, como se explicó anteriormente, y para el medio cilindro se realizó una textura que simulaba ser un techo de acero y láminas marrón. El procedimiento para realizarla fue similar al seguido para desarrollar las ventanas del proyecto del laboratorio CDM descrito anteriormente.

El estacionamiento es una cuadra sencilla con una textura desarrollada por un colega.

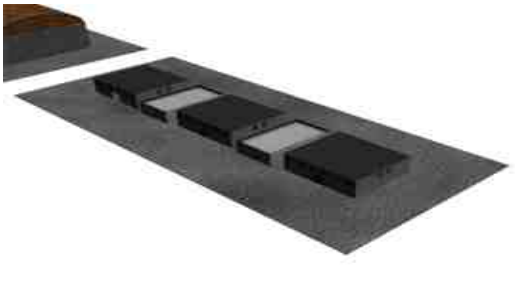


Fig. 2.44 y 2.45: Agencia y estacionamiento.

Una vez listos estos modelos, se tomaron aquellos desarrollados por el resto del equipo y se juntaron. Entonces, el proyecto completo fue importado a Unity, donde el equipo de programación se encargó de darle el aspecto óptimo y comenzar con el script.

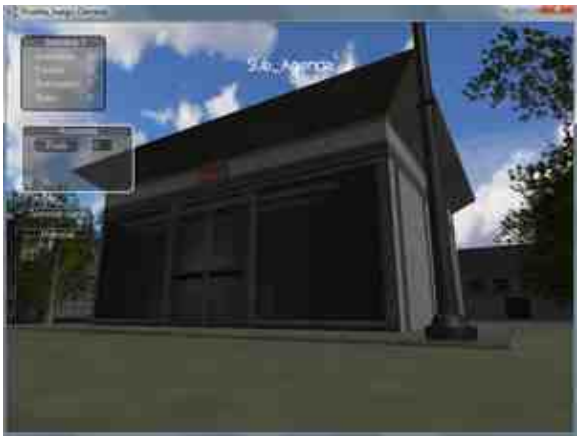


Fig. 2.46: Juego de Coca-Cola en Unity.