Desarrollo de una libreria para mapping de proyeccion usando `Processing`

Juan Carlos Rodriguez Guacaneme
Carlos Eduardo Martin Rubiano

Asesor Camilo Rey

6 de agosto de 2014

0.1. Resumen

El mapping es una tecnología de proyección utilizada para convertir objetos de formato irregular en una superficie de proyección. Utilizando un software especializado, un objeto de dos o tres dimensiones es especialmente trazado en un entorno virtual. Empleando esta información, el software puede interactuar con un proyector para adaptar una determinada imagen u animación a la superficie de un objeto. Con esta técnica se puede crear ilusiones ópticas, sensación de movimiento a los objetos previamente estáticos. El video es comúnmente combinado con audio para crear una narrativa audiovisual.[17].

Para el desarrollo se crearon dos ventanas, principal y secundaria. La ventana principal, contiene la imagen o animación, que va ser la textura a proyectar en la ventana secundaria además por medio de sus vértices, se puede reflejar una región determinada de la textura. Y la ventana secundaria contiene cada uno de los cuadriláteros, puestos en escena, estos se pueden deformar, desde cada uno de los vértices, con el fin de poderlo adaptar sobre el contorno de cualquier figura. Estructuralmente, se compone: clase cuadrilátero, incorpora todas las propiedades, y vértices de textura. vistaSecundaria PFrame, que contiene cada uno de los cuadriláteros puestos en escena, y GraphicsMapping que es la ventana principal.

Índice general

1. Preliminares **5**
- 1.1. Álgebra Lineal . . . 5
  - 1.1.1. Puntos y Vectores . . . 5
- 1.2. Sistemas de coordenadas . . . 6
- 1.3. Transformaciones Lineales . . . 6
  - 1.3.1. Propiedades Básicas de una transformación lineal . . . 7
  - 1.3.2. Transformaciones geometricas . . . 8
  - 1.3.3. Coordenadas homogéneas . . . 10
- 1.4. Algunos algoritmos de Geometría Computacional . . . 11
  - 1.4.1. Determinar si un punto está dentro de un triángulo . . . 11

2. Desarrollo de una libreria que permite hacer Mapping de proyeccion **15**
- 2.1. Mapping de proyección . . . 15
- 2.2. Proyecciones en computación gráfica. . . . 18
  - 2.2.1. El modelo de la camara. . . . 18
  - 2.2.2. Modos de proyección. . . . 18
- 2.3. Textura. . . . 21
  - 2.3.1. Coordenadas de textura. . . . 21
  - 2.3.2. Vértices de textura. . . . 21
- 2.4. Empresas dedicadas al entorno audivisual . . . 22

3. Resultados y conclusiones **25**
- 3.1. Objetivos . . . 25
  - 3.1.1. Objetivo General . . . 25
  - 3.1.2. Objetivos Específicos . . . 25
  - 3.1.3. Requerimientos de la librería . . . 26
- 3.2. Resultados . . . 26
  - 3.2.1. Especificación de casos de uso . . . 26
  - 3.2.2. Diseño UML . . . 30
  - 3.2.3. Diagrama de Clases . . . 31
- 3.3. Arquitectura de la libreria . . . 31
  - 3.3.1. Componentes . . . 31
- 3.4. PGraphics . . . 34
  - 3.4.1. Algoritmos implementados . . . 34
- 3.5. Uso de la librería . . . 36
  - 3.5.1. Ejemplos . . . 37

3.6. Conclusiones 37
3.7. Trabajo futuro 38

Capítulo 1

Preliminares

En este capítulo vamos a ver los conceptos y propiedades de Algebra lineal, Sistemas de coordenada, transformaciones lineales y geométricas, que nos permitirán construir los algoritmos y los procesos matemáticos necesarios para la librería que permite hacer mapping de proyección.

1.1. Álgebra Lineal

El algebra lineal es la rama de las matemáticas con el fin de estudiar los vectores y espacios vectoriales. Ha cobrado una gran importancia en el mundo de la computación y en particular en el campo de la computación gráfica, pues el álgebra lineal posee estructuras y conceptos que permiten, por ejemplo el manejo de imágenes, la noción de proyección, la transformación geométrica de objetos y su ubicación en el espacio n-dimensional.

1.1.1. Puntos y Vectores

En esta subsección, definiremos los *vectores* como n-tuplas de números reales que componen los puntos del espacio $\mathbb{R}^n$, junto con sus principales operaciones y su interpretación geométrica.

Definición 1.1.1 (Espacio n-dimensional)**.** *En el espacio n-dimensional está dada por una n-tupla* $\mathbf{p} \in \mathbb{R}^n$*, donde cada coordenada es un escalar. A saber,*

$$\mathbf{p} = (x_1, x_2, \ldots, x_n) \tag{1.1.1}$$

con $x_i \in \mathbb{R}$.

Un vector matemáticamente es una lista de números o arreglos de números. En computación grafica lo principal acerca de los vectores es entender cómo funciona la geometría de éstos. Un vector en términos geométrico puede ser tanto un punto como una línea apuntando hacia alguna dirección. Esta línea posee una magnitud y una dirección [13].

Así, se define la*magnitud* o **Longitud** de un vector como la longitud de sus representaciones y su dirección como la dirección de cualquiera de sus representaciones. A saber si $\mathbf{p}$ es

un vector en $\mathbb{R}^n$, su *magnitud* $||\mathbf{p}||$ se define como

$$||\mathbf{p}|| = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x_1^2 + x_2^2 + \cdots + x_{n-1}^2 + x_n^2} \tag{1.1.2}$$

Estos vectores conforman un espacio vectorial donde se le pueden aplicar operaciones algebraicas; Sus operaciones son las siguientes:

- Suma de vectores:
$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 + y_1 \\ \vdots \\ x_n + y_n \end{bmatrix}$$
- Producto escalar:
$$\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1 \cdot y_1 \\ \vdots \\ x_n \cdot y_n \end{bmatrix}$$

Para calcular la distancia entre vectores se aplica la formula de la magnitud ya que un vector se puede considerar como un segmento de una linea recta. A saber, podemos determinar que la distancia entre dos puntos por medio de

$$d(\mathbf{v}, \mathbf{w}) = ||\mathbf{v} - \mathbf{w}|| = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2} \tag{1.1.3}$$

1.2. Sistemas de coordenadas

$\mathbb{R}^n$ compone un espacio vectorial[13]. El punto $\mathbf{o} = (0, \ldots, 0)$ se conoce como el *origen*. Note que si $\mathbf{v} = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} \in \mathbb{R}^n$,podemos escribir, $\mathbf{v} = x_1 \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} + x_2 \begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{pmatrix} + \ldots + x_n \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{pmatrix}$.

Si definimos $\hat{e_1} = \begin{pmatrix} 1 \\ 0 \\ \ldots \\ 0 \end{pmatrix}$, $\hat{e_2} = \begin{pmatrix} 0 \\ 1 \\ \ldots \\ 0 \end{pmatrix}$, $\ldots$, $\hat{e_n} = \begin{pmatrix} 0 \\ 0 \\ \ldots \\ 1 \end{pmatrix}$ decimos que $\hat{e_i}$ compone una *base* de $\mathbb{R}^n$. En el álgebra lineal, esta base se conoce como el sistema de *coordenadas rectangulares*. Este sistema compone un sistema de referencia que se utiliza para localizar y colocar un punto concreto en un espacio determinado. En $\mathbb{R}^3$ se conoce como el sistema de referencia x, y y z[13].

1.3. Transformaciones Lineales

Las transformaciones lineales son parte fundamental de este proyecto, ya que tomando como base el algebra lineal estas tienen reglas que me permite cambiar el tamaño o la dirección

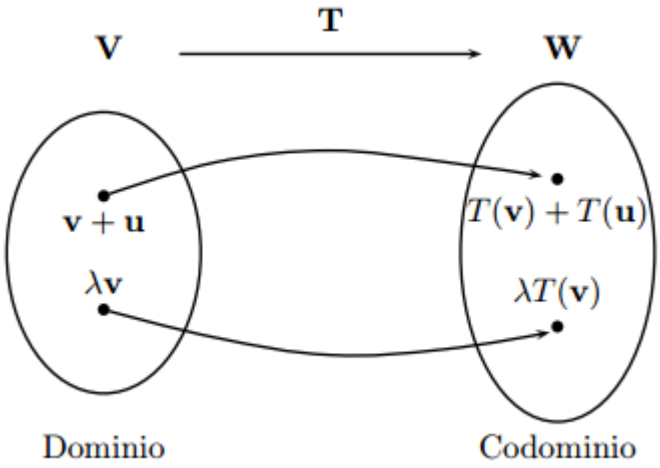


Figura 1.1: Ejemplo de una transformacion lineal (Definición 1.3.1), tomado de: [25]

de un vector y que al aplicarlas a la computación grafica permiten manipular los vectores en un plano.

Por definición una transformación lineal es una función de dos espacios vectoriales. V y W, en efecto $T : V \to W$, es una transformación lineal de V en W, si y solo sí. Se cumple estas dos propiedades:

- (Propiedad aditiva)

$$T(\mathbf{v}_1 + \mathbf{v}_2) = T(\mathbf{v}_1) + T(\mathbf{v}_2), \forall \mathbf{v}_1, \mathbf{v}_2 \in V \tag{1.3.1}$$

- (Propiedad homogenea)

$$T(\lambda \mathbf{v}) = \lambda T(\mathbf{v}) \forall \mathbf{v} \in V, \forall \lambda \in R \tag{1.3.2}$$

$$T(0) = 0 \tag{1.3.3}$$

Aquí se está denotando la suma y el producto por escalar definidos tanto en el espacio vectorial V como en el espacio vectorial W.[13]

1.3.1. Propiedades Básicas de una transformación lineal

Las propiedades de transformación lineal nos permiten demostrar que la imagen de un vector a través de una transformación es combinación lineal de las imágenes de los vectores base del espacio, usando los mismos coeficientes del vector original a transformar[13].

Teorema 1.3.1 (Igualdad entre transformaciones lineales)**.** *Sean V y W espacios vectoriales de dimensiones n y m respectivamente. Sean $T : V \longrightarrow W$ y $S : V \longrightarrow W$ transformaciones lineales de V en W. decimos que T y S son* iguales *como transformaciones lineales si y solo si*

$$S(\mathbf{v}_i) = T(\mathbf{v}_i), i = 1, \ldots, n \tag{1.3.4}$$

con $\vec{v_i}$ cualquier base de $\mathbb{R}^n$. De este resultado, se tiene trivialmente una transformación lineal que asigna al vector cero del dominio, el vector cero del codominio[13].

Corolario 1.3.2. *Sean $T : V \longrightarrow W$ una transformación lineal, entonces*

$$T(0) = 0 \tag{1.3.5}$$

Para el caso de las transformaciones lineales de $\mathbb{R}^n$ a $\mathbb{R}^m$, las rectas son enviadas en rectas o puntos y los planos son enviados en planos, rectas o puntos. Adicionalmente, el teorema 1.3.1 nos permite demostrar que una transformación lineal asigna un subespacio de dimensión k del dominio, un subespacio del codominio de dimensión o menor a k(dos funciones definidas sobre un mismo dominio y codominio son iguales, si y solo si, tienen las mismas imágenes para todos y cada uno de los elementos del dominio).[13]

1.3.2. Transformaciones geometricas

En las trasformaciones geométricas nos permiten crear o modificar una nueva figura a partir de una previamente dada. Cabe decir que a diferencia de una transformación lineal, estas permiten tratar cada una de las figuras en el espacio y linealmente se encargan de estudiar las funciones a la que puedan pertenecer[4].

Estas transformaciones se clasifican en:

- **Directa**:La nueva figura conserva el sentido original del plano cartesiano.
- **Inversa**:El sentido de la nueva figura y del original son contrarios.

Para el caso en 2 dimensiones, esto es en $\mathbb{R}^2$, consideremos el *plano cartesiano.*

Definición 1.3.3 (Plano Cartesiano)**.** *Plano cartesiano, un plano cartesiano está formado por dos rectas numéricas perpendiculares, una horizontal y otra vertical que se cortan en un punto.[4]*

Dado que el mapping juega con espacios dos dimensionales que se superponen para dar la ilusión de tridimensionalidad, necesitamos presentar las transformaciones básicas para hcaer mapping:

1. **Traslación**: cambios en la posición.
2. **Escalado**: Cambios en el tamaño.

La traslación no compone una transformación lineal *per-se* pues el vector cero no necesariamente va al vector cero. Sin embargo, es importante para los objetivos de este proyecto y la presentaremos como una transformación *geométrica* pero no necesariamente lineal.

La traslación permite reposicionar un objeto desplazándolo a las nuevas coordenadas, por medio de una ecuación de la siguiente forma:

$$\begin{aligned} x' &= x + t_x \\ y' &= y + t_y \end{aligned} \tag{1.3.6}$$

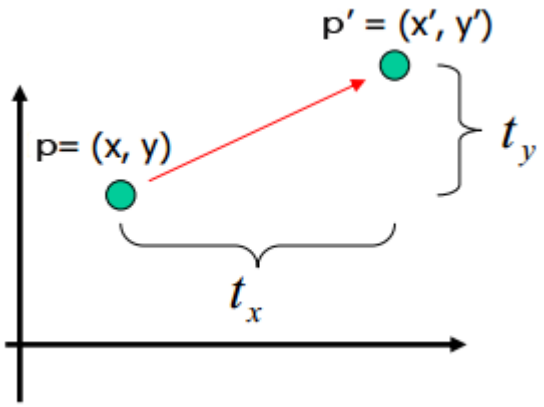


Figura 1.2: Ejemplo de traslación matricial, tomado de[21]

Vectorialmente hablando si $\mathbf{t} \in \mathbb{R}^2$,

$$\mathbf{p}' = \mathbf{p} + \mathbf{t} \tag{1.3.7}$$

donde $\mathbf{t}$ es el *vector de desplazamiento* Esta transformación no *deforma* los objetos ya que para trasladar un cuadrilátero o un polígono solo hay que trasladar solo sus vértices. El *escalado* permite hacer a los objetos más grandes o más pequeños con relación a lo horizontal y a lo vertical (independientemente). Para esta transformación, hay que especificar dos factores de escala, $s_x, s_y \in \mathbb{R}$ para cambiar el tamaño horizontal y vertical de los objetos:

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \end{aligned} \tag{1.3.8}$$

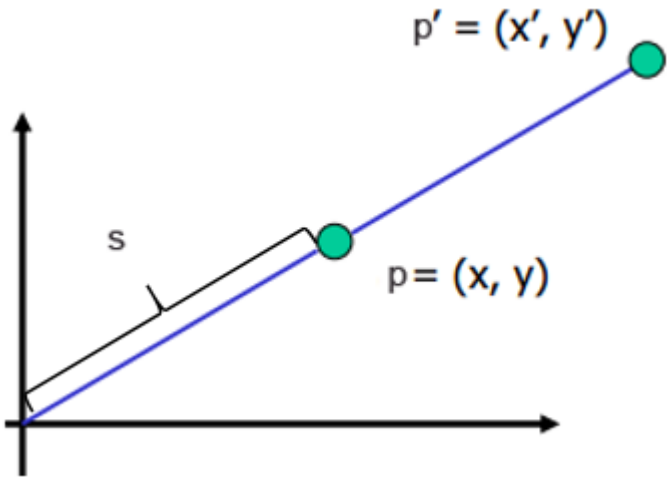


Figura 1.3: Ejemplo de escalado matricial,tomado de[21]

A saber, el escalado puede *aumentar* o disminuir el tamaño de los objetos dependiendo del valor que tomen los factores de escala s:

- si $s > 1$; aumenta su tamaño.
- si $s == 1$; no cambia su tamaño.
- si $s < 1$; disminuye su tamaño.

La *rotación* permite girar un punto o los vértices de una figura geométrica alrededor del punto **0** (el origen), dentro de un sistema de coordenadas, usando el esquema de rotación en coordenadas polares:

$$\begin{aligned} x &= R\cos(\alpha) \\ y &= R\sin(\alpha) \end{aligned} \tag{1.3.9}$$

A partir de este esquema es posible deducir una fórmula cerrada para la rotación de un punto (x, y) por un ángulo α dentro del sistema de coordenadas usual (véase 1.3.2

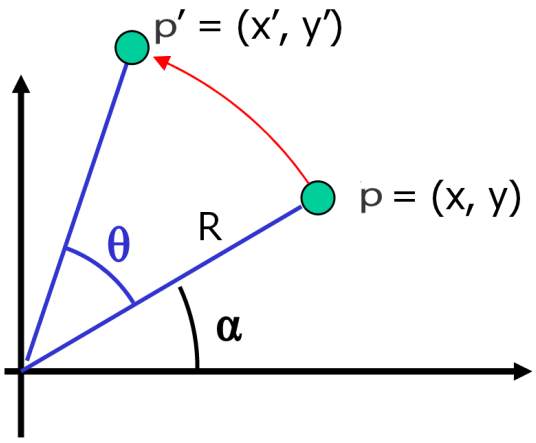


Figura 1.4: Ejemplo de rotación matricial,tomado de[21]

1.3.3. Coordenadas homogéneas

Las coordenadas homogéneas permiten tratar las transformaciones geométricas como una multiplicación de matrices, independientemente de si la transformación compone una transformación lineal[15].

Mediante la adición de una coordenada -por lo general es 1- para incluir factores de que las matrices den la posibilidad de hacer traslación y escalado.[15] Para ello se debe representar un punto cartesiano de la forma (x, y) en coordenadas homogéneas como (x, y, c). Cabe anotar que un mismo punto tiene infinitas representaciones (un punto (a, b, c) en coordenadas homogéneas representa al punto $(a/c, b/c)$) sin embargo, al escoger $c = 1$ la representación se vuelve unívoca[21]

Por medio de estas coordenadas homogéneas, la traslación se puede representar en $\mathbb{R}^2$ por medio de una matriz 3×3 por medio de la siguiente fórmula

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1.3.10}$$

en consonancia con la ecuación 1.3.6. A su vez, tanto el escalado como la rotación se pueden representar usando matrices 3×3:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1.3.11}$$

A saber si se quiere rotar un vector por un ángulo θ, la matriz apropiada es la matriz [13]

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} cos\theta & sin\theta & 0 \\ -sin\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \tag{1.3.12}$$

con $\theta \in \mathbb{R}$

1.4. Algunos algoritmos de Geometría Computacional

Con miras a la construcción de una librería para hacer mapping de proyección, es necesario incluir en el proyecto algunos algoritmos para el posicionamiento y manipulación de cuadriláteros en el plano.

1.4.1. Determinar si un punto está dentro de un triángulo

Para la selección de un cuadrilátero en el espacio, usaremos el algoritmo de verificación de un punto dentro del triángulo denominado la *Técnica Baricentrica*, que se basa en determinar la posición de un punto dentro de un triángulo con vértices $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2 \in \mathbb{R}^2$[12]. Sean $\mathbf{p}_1$, $\mathbf{p}_2$ y $\mathbf{p}_3$ los vértices del triángulo y sea $\mathbf{p} \in \mathbb{R}^2$ un punto arbitrario. Queremos determinar si $\mathbf{p}$ pertence al triangulo utilizando las coordenadas homogéneas. Veáse la figura **??**

Defina

$$\begin{aligned} \mathbf{s}_1 &= \mathbf{p}_2 - \mathbf{p}_1 \\ \mathbf{s}_2 &= \mathbf{p}_3 - \mathbf{p}_1 \end{aligned} \tag{1.4.1}$$

Entonces si $\mathbf{p}$ pertence al triángulo conformado por $\mathbf{p}_1$,$\mathbf{p}_2$ y $\mathbf{p}_3$ debemos tener que

$$\mathbf{p} = \mathbf{p}_1 + u\mathbf{s}_1 + v\mathbf{s}_2 \tag{1.4.2}$$

con $u, v \in [0, 1]$. Defina

$$\mathbf{s} = \mathbf{p} - \mathbf{p}_1 \tag{1.4.3}$$

con lo cual tenemos que

$$\mathbf{s} = u\mathbf{s}_1 + v\mathbf{s}_2 \tag{1.4.4}$$

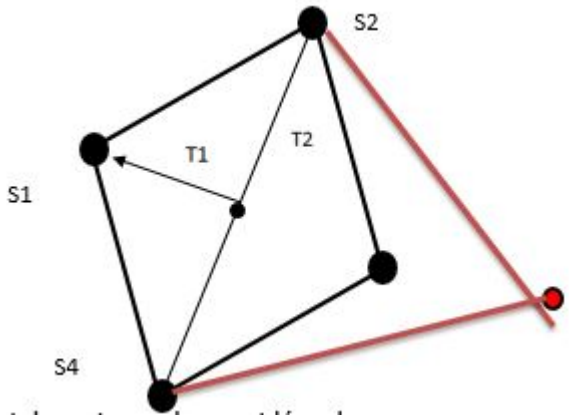


Figura 1.5: Punto dentro del triangulo.

de modo tal que

$$\mathbf{s} \cdot \mathbf{s}_1 = u||\mathbf{s}_1||^2 + v\mathbf{s}_1 \cdot \mathbf{s}_2 \tag{1.4.5}$$

y

$$\mathbf{s} \cdot \mathbf{s}_2 = u\mathbf{s}_1 \cdot \mathbf{s}_2 + v||\mathbf{s}_2||^2 \tag{1.4.6}$$

que compone un sistema de ecuaciones lineales 2×2 en términos de u y v de la siguiente forma:

$$\begin{pmatrix} \mathbf{s} \cdot \mathbf{s}_1 \\ \mathbf{s} \cdot \mathbf{s}_2 \end{pmatrix} = \begin{pmatrix} ||\mathbf{s}_1||^2 & \mathbf{s}_1 \cdot \mathbf{s}_2 \\ \mathbf{s}_1 \cdot \mathbf{s}_2 & ||\mathbf{s}_2||^2 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} \tag{1.4.7}$$

que se puede resolver por Cramer [13].

$$\Delta = \begin{vmatrix} ||\mathbf{s}_1||^2 & \mathbf{s}_1 \cdot \mathbf{s}_2 \\ \mathbf{s}_1 \cdot \mathbf{s}_2 & ||\mathbf{s}_2||^2 \end{vmatrix} \tag{1.4.8}$$

En la librería, implementamos esta rutina por medio del siguiente método

```
public boolean isInsideTriangle(PVector pos) {
    PVector v0 = new PVector(vertices[2].x - vertices[3].x,
                             vertices[2].y - vertices[3].y);
    PVector v1 = new PVector(vertices[1].x - vertices[3].x,
                             vertices[1].y - vertices[3].y);
    PVector v2 = new PVector(pos.x - vertices[3].x,
                             pos.y - vertices[3].y);
   //producto escalar
    float  dot00 = v0.dot(v0);
    float  dot01 = v0.dot(v1);
    float  dot02 = v0.dot(v2);
    float  dot11 = v1.dot(v1);
    float  dot12 = v1.dot(v2);

// Compute barycentric coordinates
  float invDenom1 = 1 / (dot00 * dot11 - dot01 * dot01);
```

```
  float u_1 = (dot11 * dot02 - dot01 * dot12) * invDenom1;
  float v_1 = (dot00 * dot12 - dot01 * dot02) * invDenom1;

    // ______________________________Triangulo 2
    PVector v3 = new PVector(vertices[0].x - vertices[1].x,
                             vertices[0].y - vertices[1].y);
    PVector v4 = new PVector(vertices[2].x - vertices[1].x,
                             vertices[2].y - vertices[1].y);
    PVector v5 = new PVector(pos.x - vertices[1].x,
                             pos.y - vertices[1].y);
    //producto escalar
    float  dot33 = v3.dot(v3);
    float  dot34 = v3.dot(v4);
    float  dot35 = v3.dot(v5);
    float  dot44 = v4.dot(v4);
    float  dot45 = v4.dot(v5);

    // Compute barycentric coordinates
  float invDenom2 = 1 / (dot33 * dot44 - dot34 * dot34);
  float u_2 = (dot44 * dot35 - dot34 * dot45) * invDenom2;
  float v_2 = (dot33 * dot45 - dot34 * dot35) * invDenom2;

   // Verificar si el punto esta dentro en uno
   //de los dos  triangulos que forman el cuadrilatero
   return ((u_1 >= 0) && (v_1 >= 0) && (u_1 + v_1 < 1)
           ||(u_2 >= 0) && (v_2 >= 0) && (u_2 + v_2 < 1));
}
```

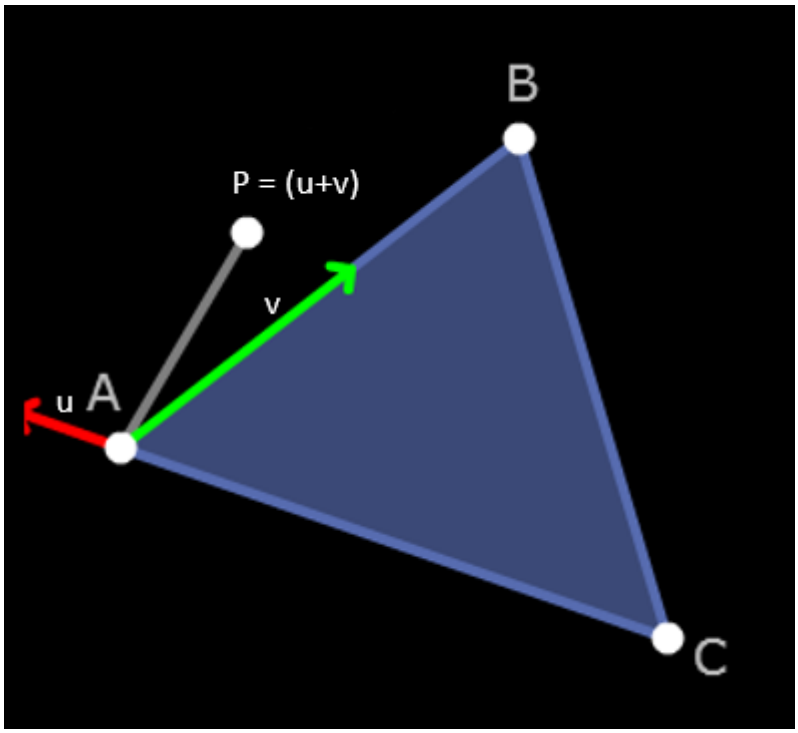


Figura 1.6: Punto en el triangulo, tomado de: [6]

Capítulo 2

Desarrollo de una libreria que permite hacer Mapping de proyeccion

En este capítulo se destaca lo que significa hacer *mapping* desde un contexto gráfico, en el cual se enfoca la librería. Además se estudia, los distintos tipos de proyecciones de mapping existentes hasta el día de hoy.

En el mundo actual, hay gran variedad de técnicas de mapping (efecto de poder reflejar o plasmar representaciones graficas), sobre objetos tridimensionales en determinadas superficies o espacios, el mapping pretende crear efectos, animaciones, videos e imágenes sobre superficies no regulares. Entre las aplicaciones de éste queremos citar en particular, la proyección sobre monumentos como: Castillo Gorey, Presidencia Española de la UE, Teatro Colon, etc[10].

Se encuentran interactuando o reflejando gran variedad de proyecciones sobre las fachadas de estos monumentos privados o gubernamentales, que combinaron tecnología y arte, se logra enfocar ante el ojo humano un sentido más de realidad de la imagen que se proyecta, y a la vez se vuelve muy iterativo con el entorno real. Cabe decir que el concepto del mapping empieza a ser usado desde el año 1969 en Disneyland, donde por medio de cabezas sin cuerpo, iluminaban la mansión embrujada. Posteriormente en 1980 Michael Naimark uso filmaciones de personas sobre una sala, simulando que interactuaban con los objetos. Desde este momento ya se empieza a estudiar el uso de proyecciones y surge el concepto de mapping. Hasta el día de hoy que se ha logrado fortalecer en gran medida.[7]

2.1. Mapping de proyección

El mapping es una adpatación de la noción de proyección utilizada para convertir objetos con superficies irregulares en el objeto de proyecciones de forma tal que se vean tridimensionales[17]. La proyección sobre superficices irregulares se obtiene utilizando software especializado para fraccionar la proyección sobre una colección de objetos de dos o tres dimensiones que definen lo que se conoce como la *escena*.

Empleando esta información, el software puede interactuar con un proyector para adaptar

(a) (b)

Figura 2.1: Dos ejemplos de tecnicas de video mapping. (a) Una proyección audivisual video Mapping sobre un edificio (b) Un video Mapping sobre la fachada del edificio del reloj de la puerta del sol. Tomado de. [14]

una determinada imagen u animación a la superficie de un objeto. Con esta técnica se puede crear ilusiones ópticas, sensación de movimiento a los objetos previamente estáticos. El video es comúnmente combinado con audio para crear una narrativa audiovisual. Para poder hacer mapping se debe tener en cuenta los siguientes elementos básicos:

1. Sistema de proyección.
2. Superficie de proyección.
3. Punto de visualización.

Un sistema de proyección consta de diferentes características que influyen de forma directa en la proyección final

- Sistema Óptico.
- Luminosidad.
- Contraste.

Influye de forma directa en el proceso de calibración del mapeado, ya que determina el frustum de proyección. **Definición:** Frustum, Un frustum es un cono de proyección.

- **Cociente**: indica la apertura del sistema de proyección. Ej. Un cociente 1:1 permite disponer de una proyección de 1m de ancho a 1m de distancia, mientras que un cociente 1:2 crea una imagen de 1m de ancho a 2m de distancia.
- **Desplazamiento**: desplazamiento del eje óptico. Necesario para colocar el proyector.
- **Distancia focal**: representa la longitud del sistema óptico. Una focal corta permite una gran apertura, pero aumenta la profundidad de campo, lo cual puede provocar problemas de enfoque cuando se proyecta en objetos tridimensionales.

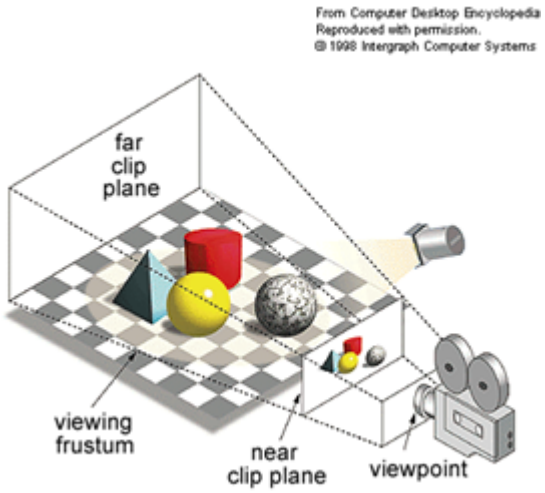


Figura 2.2: frustum, tomado de:[5]

Con los parámetros cociente, relación de aspecto y desplazamiento, podemos calcular el frustum del sistema de proyección y proyectar sobre diferentes superficies. A su vez, las superficies son el espacio donde se da vida al mapping tiene importancia y se clasifican en distintos tipos:

- Superficies planas, como paredes, suelos, techos, etc.
- Superficies tridimensionales:
  - **Continuas:** Superficies curvas, Cúpulas, esferas, columnas, etc.
  - **Discontinuas:** Diferentes planos de proyección. Composiciones geométricas, edificios.

En efecto, el video mapping pretende ser una proyección sobre una superficie no plana, en la cual se adapta el contenido a proyectar sobre la superficie de proyección. Para esto, es necesario un *estudio previo* de la superficie de proyección y su descomposición en múltiples planos sobre los cuales se proyecta contenido multimedia (objetivo principal de la librería desarrollada en este trabajo).

En el mapping se puede dar distintos puntos de visualización dependiendo del número de proyectores que se tiene apuntando a una superficie específica, para combinar diferentes proyecciones y hacer más viva la ilusión de la proyección tridimensional. En efecto, los dos factore que hacen efectivo un mapping son:

- El punto de vista del usuario que determina la percepción del video mapping y permite producir la ilusión de lo tridimensional en la proyección.
- La dirección de los proyectores sobre la escena a proyectar. De no estar en armonía con la visión de los espectadores, el efecto 3D puede no ser coherente.[5]

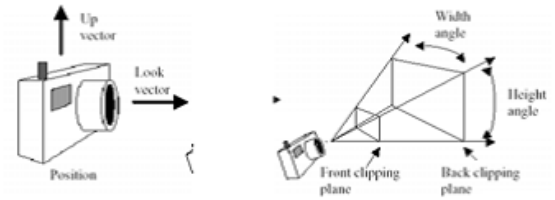


Figura 2.3: Posición de la cámara.

2.2. Proyecciones en computación gráfica.

En la computación grafica se generan proyecciones tanto en $2D$ o $3D$, donde el ojo humano puede ver una proyección de algún objeto del mundo real a una proyección realizada por un computador.Existen dos tipos de proyección. La proyección ortogonal es la encargada de mapear los objetos directamente en la pantalla sin alterar su integridad, depende de un plano de proyección y el tamaño de los objetos es invariante. La proyección perspectiva se ven representados los objetos como en la vida real, es decir que entre más cerca estén los objetos del punto de vista más grande se ven, y si están más lejos más pequeño se ven.[18]

2.2.1. El modelo de la camara.

.

Un modelo de cámara debe tener información como: Posición, Orientación, campo de visión, proyección perspectiva u ortográfica(cámara cerca de los objetos o a distancia infinita). Esto lo podemos ver en la sección: **??** y 2.2.2

La orientación de la cámara está determinada por el vector `lookAt` y el ángulo en el que está rotada la cámara con respecto a ese vector, es decir, la dirección del vector `up` [14].

La posición de la cámara es un punto en el espacio tridimensional (x, y, z)

2.2.2. Modos de proyección.

El proceso de visión en tres dimensiones es más complejo en dos dimensiones, en dos dimensiones solamente se necesita definir una ventana y un marco mientras que en tres dimensiones se debe realizar una transformación de tres a dos dimensiones llamada proyección. Existen dos tipos de proyección capaces de generar vistas de los objetos tridimensionales:

1. La proyección *ortográfica*
2. La proyección *en perspectiva*

La proyección ortográfica es utilizada para generar diferentes vistas de los objetos respetando sus medidas reales. Ésta proyección es de gran utilidad en arquitectura y diseño y de la cual surgen las clásicas vistas superior, frontal y lateral,que se muestran en la figura.**??** Una proyección ortográfica superior se denomina vista de planta, mientras que las proyecciones

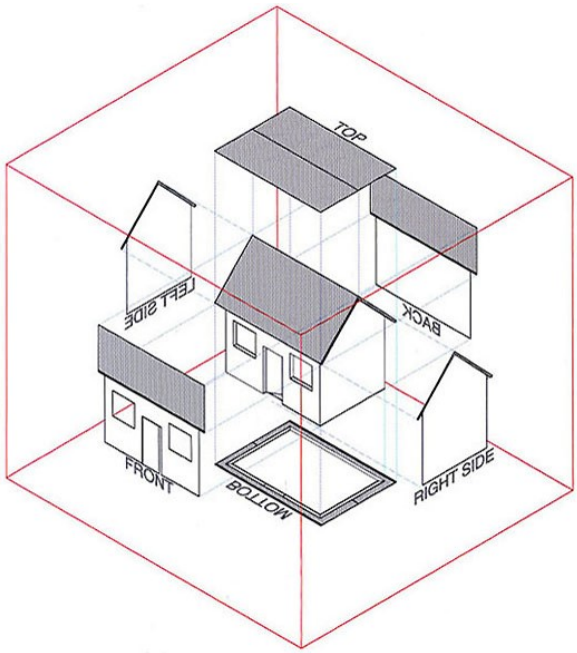


Figura 2.4: modo ortográfico

ortográficas frontal, lateral y posterior reciben el nombre de elevaciones. La *definición* de la proyección se define como la elevación correspondiente a lo que vería un observador. Esto es, es una forma más clara de visualizar una vista principal de una pieza. Es importante tener en cuenta que este modo de proyección depende de un plano de proyección y el tamaño de los objetos es invariante.

La *proyección de perspectiva*, los objetos en un plano de visión se transforman a lo largo de líneas que convergen en un punto denominado centro de referencia de proyección, ya que dependen de un *punto de fuga*. A partir de la orientación del plano de proyección se controla el número de puntos de fuga como proyecciones uno, dos y tres como se muestra en las figuras **??** y 2.2.2.

Como vemos en las figuras **??** y 2.2.2, la proyección en perspectiva produce un cono con base poligonal -*frustum*- donde la parte superior ha sido cortada por un plano paralelo a su base.[5]. Los objetos que caen dentro del volumen de visualización se proyectan hacia el ápice del frustum, donde se encuentran la cámara o punto de visión. Esto hace que los objetos más cerca de la cámara se ven más grandes porque ocupan un gran volumen de visualización a los más alejados[5].

Es por esto, que este tipo de proyección nos proporciona un grado de realismo, puesto que es similar a la forma en que trabaja el ojo humano.

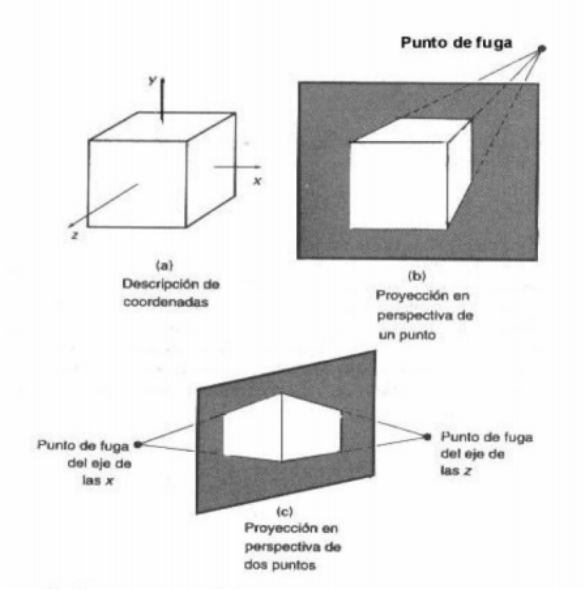


Figura 2.5: puntos de fuga en la proyección perspectiva

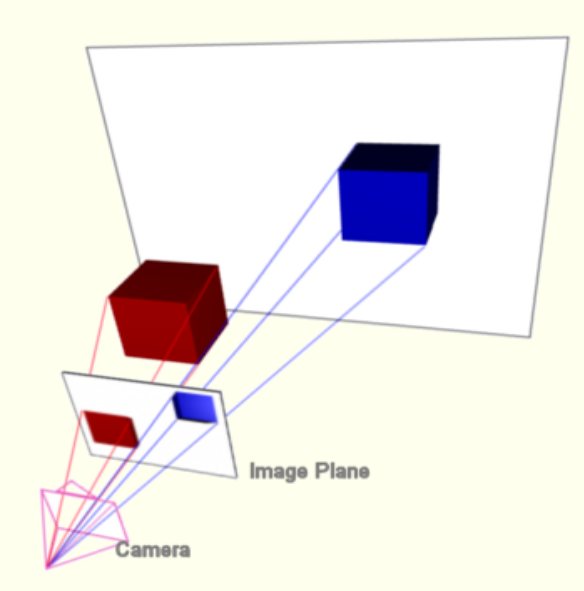


Figura 2.6: Proyección de perspectiva

2.3. Textura.

Una textura es una imagen que puede ser aplicada sobre una superficie en $3D$. Los vértices de una cara se asocian a partes de la imagen a través de las coordenadas de textura. Generalmente para mapear una superficie $2D$ en una $3D$, se toman los vértices del polígono representando la superficie y se le asocian los **texels**(Es la unidad mínima de una textura aplicada a una superficie) de la imagen. Se pueden tener normalizadas o sobre el tamaño de la imagen, con lo cual los intermedios se obtienen por interpolación.

A saber, una superficie en el espacio se puede describir como una función paramétrica vectorial

$$\mathbf{S} : \mathbb{R}^2 \to \mathbb{R}^3 \tag{2.3.1}$$

donde el dominio en $\mathbb{R}^2$ de la superficie se conoce como el *dominio paramétrico* se toma usualmente como un rectángulo en $\mathbb{R}^2$, que es posible rectificar hasta el cuadrado unitario $[0, 1] \times [0, 1]$[16].

Al rasterizar una superficie para visualización en dos dimensiones, el color asociado a un píxel se puede definir de múltiples formas: color plano, sombreado por medio de iluminación local o global ó *texturizado* utilizando una imagen.

2.3.1. Coordenadas de textura.

A la hora de texturizar una superficie paramétrica, los puntos del dominio paramétrico de la superficie se pueden asociar con posiciones dentro de la imagen de textura. Estas posiciones se conocen como *Texels* o *coordenadas UV*.

La interpolación de la textura se puede hacer con tres modos como expone la figura 2.3.1.

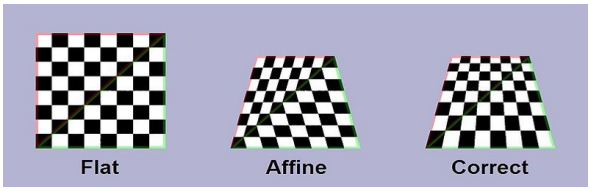


Figura 2.7: Interpolación de textura

2.3.2. Vértices de textura.

A través de la siguiente regla de tres se puede representar los vértices dentro del intervalo permitido. La figura 2.3.2, muestra los cálculos asociados a la caracterización de los vértices de textura asociables a un objeto gráfico. Las coordenadas del espacio de textura están normalizadas, independientemente del tamaño varian entre 0 y 1.

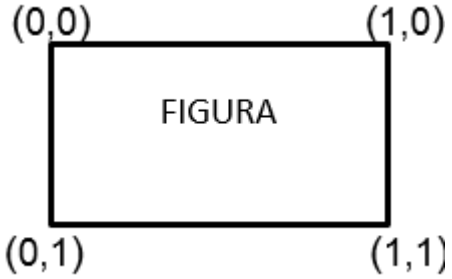


Figura 2.8: coordenada de textura

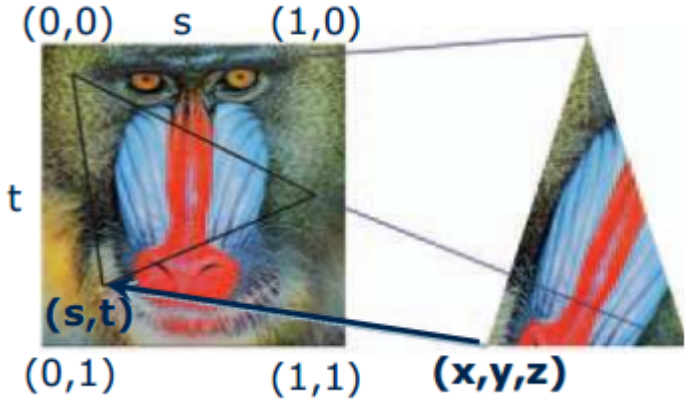


Figura 2.9: coordenada del espacio de textura, tomado de: [16]

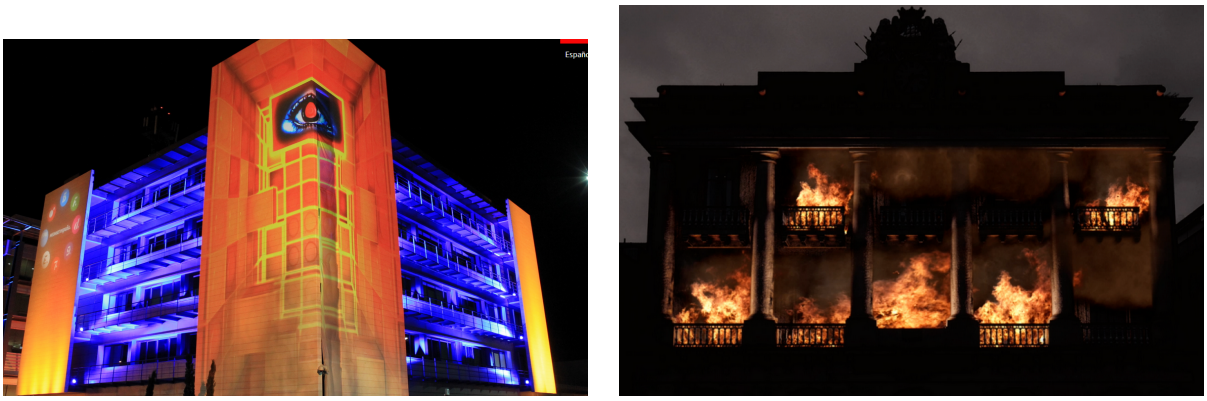
2.4. Empresas dedicadas al entorno audivisual

Hoy en día, las empresas que están enfocadas en la tecnología del mapping de proyección, están adaptando técnicas de proyección en superficies irregulares e incluso elementos tridimensionales, que permiten el desarrollo de espectáculos audiovisuales en los que es posible darle emoción a edificios y fachadas mediante la recreación de ilusiones ópticas. A continuación se mencionar algunas de las empresas que se enfocan a este espectáculo del mundo del vídeo mapping.

MAPP3D INTERACTIVE, es una empresa que lleva más de 20 años de experiencia dedicada al desarrollo de audiovisuales, producción de eventos, proyecciones $3D$ Mapping, Sistemas interactivos e iluminación espectacular[2] (véase la figura **??**.

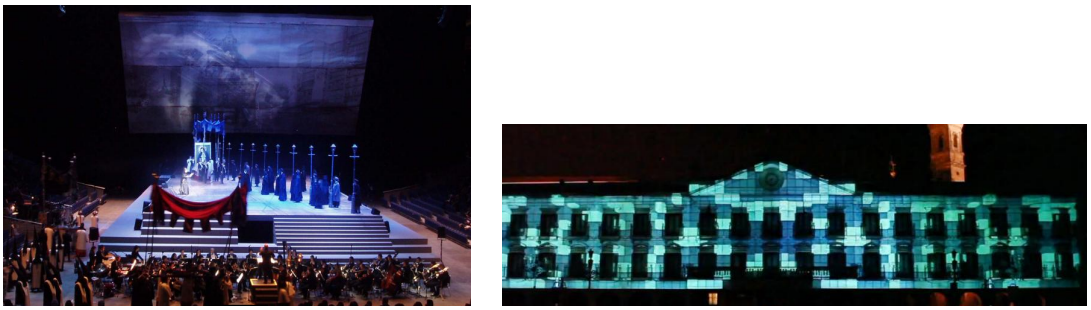
3DSCENICA, Es una empresa que parte del concepto o idea de un cliente y hace propuestas multimedia que captan la atención del público y garanticen la comunicación del mensaje e ideas del cliente. El propósito de 3DSCENICA es provocar sensaciones y emociones, estimulando los sentidos que nos relacionan con nuestro entorno [2](véase la figura 2.11.

Visualma: es una empresa mexicana de multimedia que desarrolla y explora tecnologías



(a) (b)

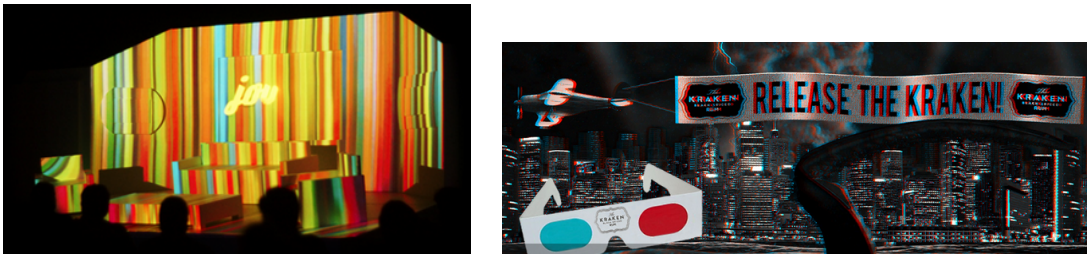
Figura 2.10: Portafolio de MAPP3D. (a) video Mapping sobre un edificio (b) video Mapping sobre la fachada del edificio.Tomado de. [2]



(a) (b)

Figura 2.11: Portafolio de 3DSCENICA. (a) Multi Proyección (b) victoria green capital. [2]

para aplicaciones en la comunicación, el espectáculo y el arte [5]. (Véase la figura **??**).



(a) (b)

Figura 2.12: Portafolio de Visualma. (a) 3D mapping para presentación de Bitcino Living Light en México. (b) Primer $3D$ mapping anaglifo en México. [5]

Capítulo 3

Resultados y conclusiones

En este capítulo se presentará la librería desarrollada con base en los conceptos expuestos en los capítulos anteriores y que constituye el producto tangible de este proyecto. Esta librería se hizo utilizando `Processing`. Un ambiente de desarrollo integrado (IDE) que facilita la generación de gráficos por computador en 2 y 3 dimensiones usando `Java` con miras a la generación de aplicaciones de código creativo de forma rápida y sencilla involucrando contenido multimedia.[22]. Los promotores y creadores de `Processing` fueron Ben Fry y Casey Reas desde el MIT-Media Lab. La herramienta es de licencia GNU, es decir a través de un grupo de personas han publicado más de 200 versiones desde el 2001 hasta el 2013[19].

`Processing` genera `Applets` que se pueden exportar para uso web, móvil y aplicaciones de escritorio. También se ha extendido para incluir elementos de Javascript y android[22]. Este proyecto quiere extender la funcionalidad de `Processing` para incluir elementos de mapping multimedia y constituir un componente nuevo.

3.1. Objetivos

Teniendo en cuenta la importancia que ha tenido el mapping de proyección, nos proponemos a realizar los siguientes objetivos, que constan en crear una librería de fácil uso, que permite realizar proyecciones sobre superficies, y dar un efecto artístico ante el ojo humano.

3.1.1. Objetivo General

Desarrollar una librería para Processing que permita hacer mapping sobre superficies irregulares.

3.1.2. Objetivos Específicos

1. Investigar acerca de los esquemas de transformación lineal para Processing.

2. Diseñar la maquinaria teorica para hacer una aplicación que haga mapping.

3. Construir la librería propuesta.

3.1.3. Requerimientos de la librería

A continuación se definen cada una de las funcionalidades que tendrá, la librería como producto final del proyecto, tanto requerimientos funcionales y no funcionales.

- **Requerimientos Funcionales:**
  1. El usuario debe generar dos ventanas principal y secundaria.
  2. El usuario permite agregar $n > 0$ cuadriláteros en la ventana secundaria.
  3. El usuario permite mapear los vértices de textura desde la ventana principal hacia el cuadrilátero seleccionado en la ventana secundaria.
  4. El usuario permite redimensionar o deformar cada uno de los cuadriláteros, moviendo y trasladando los vértices.
  5. El usuario permite mapear videos, imágenes o animaciones en cada uno de los cuadriláteros.
  6. El usuario permite desplazar los cuadriláteros , en la ventana secundaria.
  7. El usuario permite crear objetos en $2D$ para ser mapeados sobre figuras en $3D$.
  8. El usuario permite guardar y cargar los objetos mapeados.
- **Requerimientos no Funcionales:**
  1. La librería es de fácil uso.
  2. La librería es conforme al estandar de Processíng.
  3. La librería permite guardar y cargar el mapping en un archivo plano.

3.2. Resultados

3.2.1. Especificación de casos de uso

Un caso de uso es el conjunto de escenarios relacionados que describen de qué manera los actores usan el sistema para conseguir un determinado objetivo, además de la forma, tipo y orden en como los elementos interactúan y como actúa.A continuación describiremos los casos de uso que consideramos a la hora de diseñar la librería e implementarla.

Caso de uso: El usuario debe generar dos ventanas principal y secundaria.
Actores: usuario
Pre condición: Compilación IDE Processing
Post condición: Visualización de ambas ventanas principal y secundaria
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria

Caso de uso: El usuario permite agregar $n > 0$ cuadriláteros en la ventana secundaria.
Actores: usuario
Pre condición: Visualización de las dos ventanas principal y secundaria
Post condición: Cuadrilátero puesto en escena, dentro de la ventana secundaria
Escenario principal, curso lógico de sucesos:

Sistema
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. En la ventana secundaria, con la tecla 1, se puede agregar cuadriláteros dentro de la misma

Caso de uso: El usuario permite mapear los vértices de textura desde la ventana principal hacia el cuadrilátero seleccionado en la ventana secundaria.
Actores: usuario
Pre condición: Textura cargada en la ventana principal
Post condición: al agregar el cuadrilátero en la ventana secundaria, tendrá como textura, la imagen, que está cargada por defecto en la ventana principal
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. En la ventana secundaria, con la tecla 1, se puede agregar cuadriláteros dentro de la misma 4 el fondo del cuadrilátero será el mismo que contenga la ventana principal. 5.Por medio del click del mouse en la ventana principal, se puede hacer una especie de zoom, y este a su vez se ve reflejado en el cuadrilátero

Caso de uso: El usuario permite redimensionar o deformar cada uno de los cuadriláteros, desde los vértices.

Actores: usuario
Pre condición: Visualización de ambas ventanas principal y secundaria
Post condición: Con la iteración del mouse de puede redimensionar o deformar el cuadrilátero
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. En la ventana secundaria, con la tecla 1, se puede agregar cuadriláteros dentro de la misma 4. Con clic secundario sobre uno de los vértices del cuadrilátero, se puede redimensionar o deformar el mismo

Caso de uso: El usuario permite mapear videos, imágenes o animaciones en cada uno de los cuadriláteros.
Actores: usuario
Pre condición: Visualización de ambas ventanas principal y secundaria
Post condición: Dependiendo, la carga que se tiene por defecto en la ventana principal será el resultado en los cuadriláteros de la ventana secundaria.
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. En la ventana Principal, saldrá el resultado a mapear puede ser imagen, video o animación 4. Al agregar el cuadrilátero, en la ventana secundaria, tendrá de fondo la textura de la ventana principal.

Caso de uso: EL usuario permite desplazar los cuadriláteros, en distinta posición x, y de la ventana secundaria

Actores: usuario
Pre condición: Visualización de ambas ventanas principal y secundaria
Post condición: Con el botón secundario, se puede desplazar el cuadrilátero, a cualquier posición dentro de la ventana secundaria
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. Con tecla 1 se agrega un cuadrilátero a la escena 4. Con click principal sostenido, se puede desplazar el cuadrilátero a cualquier posición dentro de la misma ventana

Caso de uso: El usuario permite crea objetos en $2d$ para ser mapeados sobre figuras en $3d$
Actores: usuario
Pre condición: Visualización de ambas ventanas principal y secundaria
Post condición: El fondo, que tiene la ventana secundaria, son los objetos en $3d$ a ser mapeados
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. Con tecla 1 se agrega un cuadrilátero a la escena 4. Con base al fondo, que se tenga en la ventana secundaria, con click secundario o principal se ajusta el cuadrilátero en 2d a una parte de sección de dicha ventana, puede ser una sub figura algo con forma rectangular, con el fin de animarlo

Caso de uso: El usuario permite guardar y cargar los objetos mapeados
Actores: usuario

Pre condición: Guardar y cargar los objetos del mapeo de ambas ventanas principal y secundaria
Post condición: Con la tecla q guarda los objetos de la ventana y la tecla w carga dicho objeto
Escenario principal, curso lógico de sucesos:

usuario
1. Invocar la librería, desde cualquier IDE 2. El sistema por defecto, ejecuta dos ventanas Principal y secundaria 3. Con tecla 1 se agrega un cuadrilátero a la escena 4. Con base al fondo, que se tenga en la ventana secundaria, con click secundario o principal se ajusta el cuadrilátero en 2d a una parte de sección de dicha ventana, puede ser una sub figura algo con forma rectangular, con el fin de animarlo 4. Teniendo el mapeo, con la tecla q guardara el estado de sus objetos y con la tecla w cargara los objetos en su debida posición.

3.2.2. Diseño UML

Por medio de los diagramas `uml` queremos representar las relaciones entre los objetos que componen la librería y que implementan los algoritmos necesarios para hacer *mapping*.

El usuario que quiera invocar la librería desde `Processing` está invocando la generación de la ventana principal (el *sketch* donde está trabajando) que gestiona las texturas a a presentar y una vista secundaria (*Escenario Principal*) que le permitirá agregar n cuadriláteros a mapear `Ventana Secundaria` a los cuales se asocia una textura específica. Al agregar cuadriláteros podrá mapear sus vértices de textura desde la *Ventana Principal* hacia el cuadrilátero seleccionado en la ventana secundaria, donde el usuario al interactuar con el mouse podrá redimensionar o deformar cada uno de los cuadriláteros desde sus vértices, permitiendo mapear videos, imágenes o animaciones en cada uno de los cuadriláteros.
La libreria permite al usuario desplazar los cuadriláteros usados al presionar el botón secundario en la ventana secundaria y adaptar las coordenadas de textura usando el mouse en la ventana princial. Adicionalmente, la libreria tiene la opcion de guardar y cargar una escena completa -los objetos mapeados-, por medio de la tecla q guarda los objetos que contiene la

ventana y con la tecla w carga un archivo en format JSON con el contenido de toda una escena desarrollada previamente.

Para ver el flujo interno de llamados de la librería con relación a la interacción con el usuario final véase las diferentes figuras en 3.1 ítems (a) a (h).

3.2.3. Diagrama de Clases

El proyecto consta de las siguientes clases **Subview** (Ventana para colocar cuadriláteros), y **Parentview** (Ventana para mapear determinada sección del cuadrilátero puesto en la clase `Subview`. Posee dos ventanas una principal `Parentview` y `Subview`). Desde `Subview` se instancian la clase cuadrilátero con todas sus propiedades, dicho objeto estará compartiéndose por la clase `MappingSingleton` con el fin de actualizar y retornar, la misma lista de cuadriláteros sobre la ventana `Parentview` y `Subview`.

Para guardar el estado actual de todos los ObjetosCuadrilatero, se tiene una clase controlador `LoadJSON`, quien va tener las propiedades de serializar o deserializar la información a través de la clase `ConvertQuadrilateral`(Convertir la información de `Processing` a variables primitivas de java), Véase el diagrama de clases 3.2.

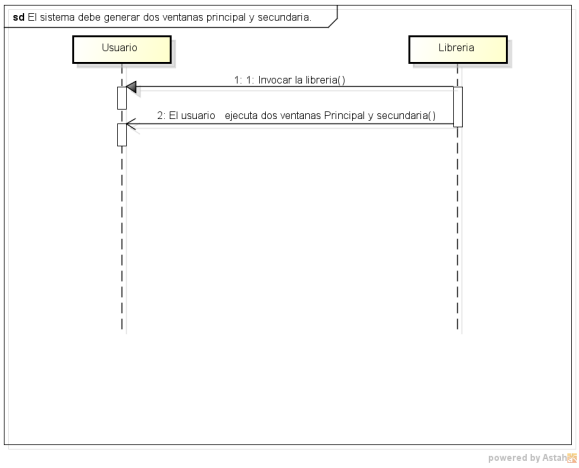
3.3. Arquitectura de la libreria

La librería se estructura con base en un patrón MVC, ya que en todo instante, el usuario puede manipular los distintos cuadrilateros puestos en la interfaz (tanto en la ventana principal como en la secundaria). El patrón MVC permite evidenciar y corregir fácilmente errores gráficos que hacen parte de la visualización que se quiere producir (el *mapping* como tal). cabe decir que este modelo es útil en el proyecto ya que por lo general el desarrollo se basa en el Core de `Processing`, tanto los métodos propios de la estructura como de las ventanas o vistas en general[22].

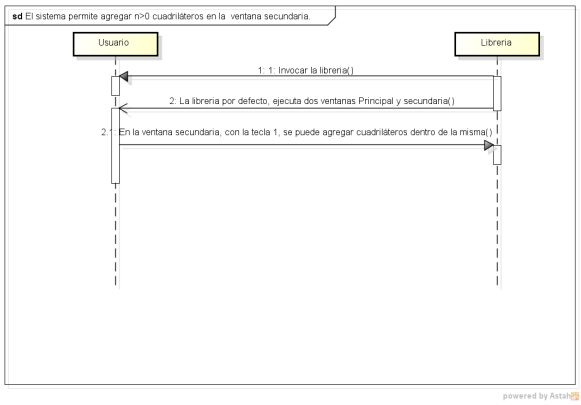
3.3.1. Componentes

Los componentes de la librería, comprenden una clase principal -un controlador- que permite gestionar la visibilidad y propiedades de cada uno de los cuadriláteros a crear, con todos sus atributos y operaciones, y es el pilar para las dos clases restantes que permiten su visualización y gestión desde están en constante referencia tanto por la ventana principal como secundaria.

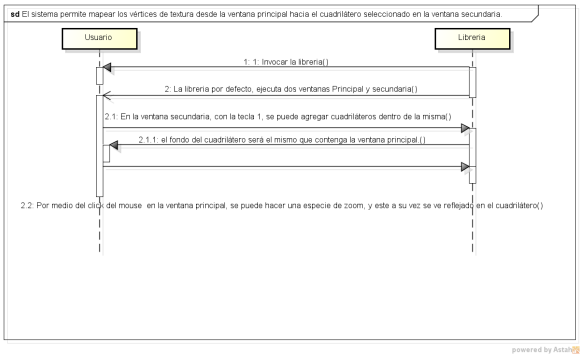
Con respecto a la la función en ambas ventanas, se utilizaron listas de objetos -como atributos de la clase controlador-que se instancian de modo tal que cada uno de los cuadriláteros que entran en escena, controlan su propio conjunto de vértices geométricos y de textura por medio de arreglos $PVector[]$. Por ende la clase principal `cuadrilatero`, está en control de todas las operaciones de desplazamiento, deformación y mapeo de textura y reaccionan a los eventos generados por el mouse.



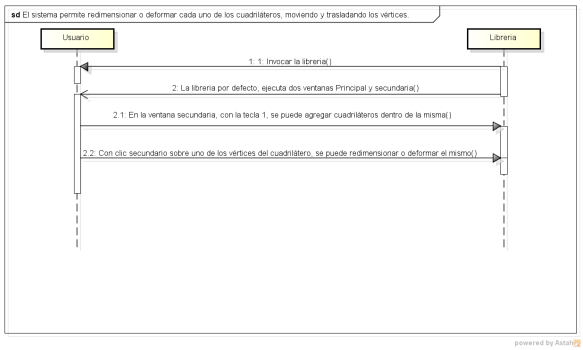
(a)



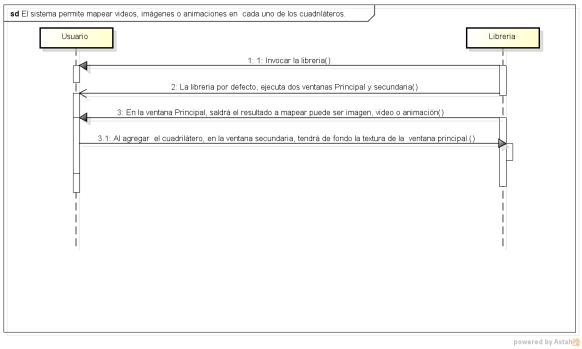
(b)



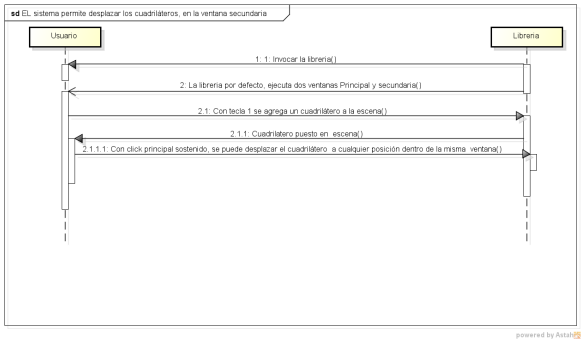
(c)



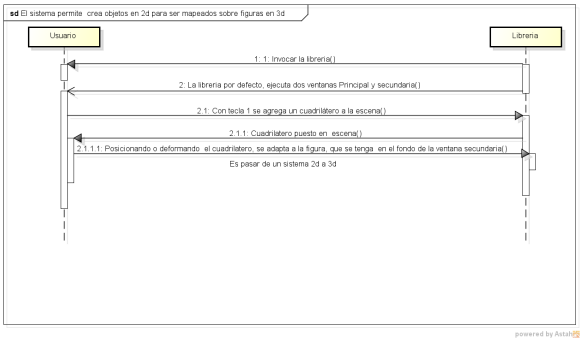
(d)



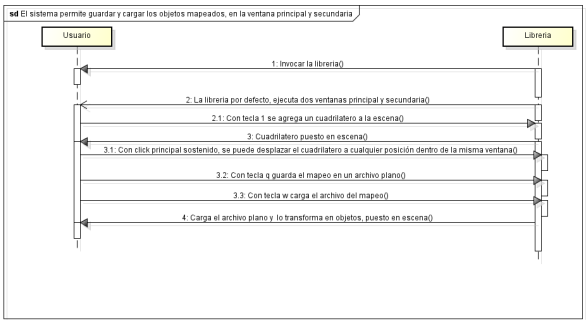
(e)



(f)



(g)



(h)

Figura 3.1: Diagramas de secuencia por requerimiento.(a) Requerimiento 1. (b) Requerimiento 2. (c) Requerimiento 3. (d) Requerimiento 4. (e) Requerimiento 5. (f) Requerimiento 6. (g) Requerimiento 7. (h) Requerimiento 8.

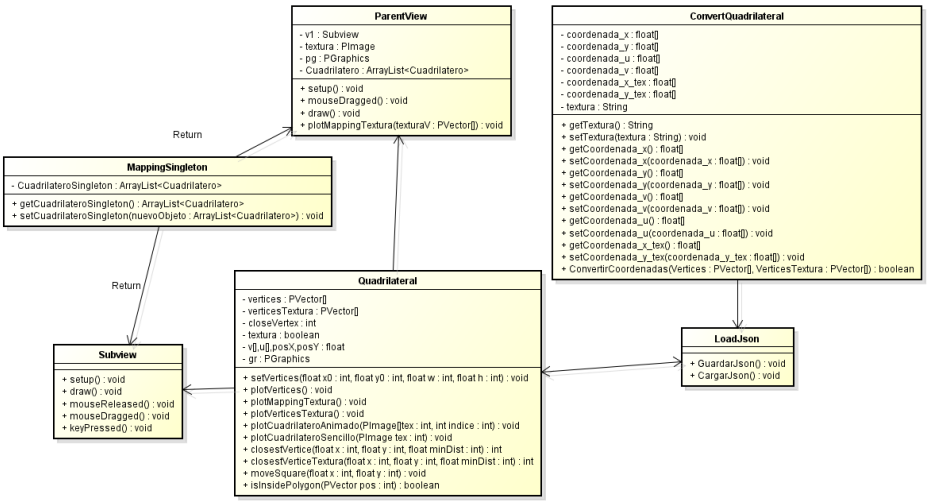


Figura 3.2: Diagrama de clases

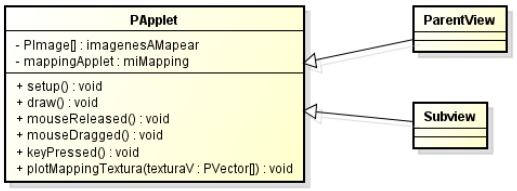


Figura 3.3: Diagrama de clases (Usuario Final)

La función de la *ventana principal* es, proveer la textura al cuadrilátero, y a su vez alterar que región se proyecta. A su vez, la *ventana secundaria* es el resultado final, se puede ver cada uno de los cuadriláteros, formando algún tipo de animación y que son la ventana a proyectar usando por ejemplo un *video-beam*. Por ultimo, cabe anotar que el diseño implementado, mantiene la integridad de cada objeto y admite potencialmente la generación de infinitos cuadriláteros para cubrir grandes superficies.

3.4. PGraphics

El `PGraphics` implementado por Core de Processing es una extensión del tipo `PImage` que permite generar animaciones en 2 y 3 dimensiones sobre el cual se puede hacer llamados a instrucciones de Processing y que se pueden asociar con periféricos. En efecto, el tipo `PGraphics` puede considerarse como un programa de `Processing` que se puede utilizar como textura sobre un objeto gráfico utilizando un bloque de dibujo

```
beginDraw()
endDraw()
```

en el contexto del bloque `beginDraw()-endDraw()`, `PGraphics` contiene todos los métodos de dibujo de objetos primitivos tanto en $2D$ como en $3D$, ya que dibujar en este lenguaje no se limita solamente a dibujar dentro de una ventana de `Processing` generando un `PApplet`, sino sobre cualquier objeto sin importar el tipo `PGraphics` como texturas. Los métodos presentados en la siguiente sección aprovechan esta clase `PGraphics` en conjunto con la facilidad de interacción con el mouse que caracteriza a `Processing` como ambiente de desarrollo.

3.4.1. Algoritmos implementados

Los métodos usados para hacer el mapeo de proyección (*mapping*) son bastante sencillos en su construcción e implementación. La visualización de los cuadriláteros y sus mapas de textura dentro de un sketch de Processing se hacen utilizando el bloque de forma `beginShape()-endShape()` propio de `Processing`. Una gran ventaja en Processing es la posibilidad de utilizar como *textura* distintos tipos de objetos: imágenes, videos y hasta objetos de tipo `PGRaphics`. El siguiente método se encarga de dibujar un cuadrilátero con su respectiva textura.

```
public void plotCuadrilateroSencillo(PImage tex) {
    gr.beginDraw();
    gr.beginShape();
    gr.texture(tex);
    gr.textureMode(NORMAL);
    gr.vertex(this.vertices[0].x, this.vertices[0].y, u[0], v[0]);
    gr.vertex(this.vertices[1].x, this.vertices[1].y, u[1], v[1]);
    gr.vertex(this.vertices[2].x, this.vertices[2].y, u[2], v[2]);
    gr.vertex(this.vertices[3].x, this.vertices[3].y, u[3], v[3]);
gr.endShape(CLOSE);
    gr.endDraw();
```

```
}
```

Naturalmente, es necesario que tanto los vértices del cuadrilátero como sus vértices de textura hayan sido inicializados. Nótese que la librería no hace validación de la correcta posición de los vértices del cuadrilátero (o sus vértices de textura), razón por la cual es responsabilidad del usuario generar cuadriláteros con cuatro vértices distintos y vértices de textura correctos.

Dado que en el *mapping* es necesario deformar los cuadriláteros para hacer que se acomoden a formas poligonales no necesariamente regulares, es preciso que un cuadrilátero se puede deformar o desplazar dentro de la ventana de proyección. Esta selección se hace utilizando un *umbral* de sensibilidad para ver cual es el vértice de un cuadrilátero más cercano a una posición (x, y) dada. El siguiente método valida esta condición y retorna el índice del vértice más cercano a la posición dada (con la sensibilidad `minDist` dada) o retorna -1 si no se encuentra ningún vértice con esta característica.

```
public int closestVertice(float x, float y, float minDist) {
    int closest =-1;
    for (int i=0;i<vertices.length;i++) {
      if (this.vertices[i].dist(new PVector(x, y))<minDist) {
        closest = i;
      }
    }
    return closest;
  }
```

el siguiente método permite trasladar todo un cuadrilátero a una posición determinada a través de la posición actual del mouse para ubicarlo en el espacio de proyección utilizando movimiento rígido (una transformación afín).

```
  public void moveSquare(float x, float y) {
    for (int i=0;i<this.vertices.length;i++) {
      this.vertices[i].x+=x;
      this.vertices[i].y+=y;
    }
  }
```

Este método es uno de los más importantes en la librería: permite saber si la posición actual del mouse está dentro o fuera del cuadrilátero para modificar tanto las coordenadas de textura en la ventana principal, como para reposicionar y mover sus vértices en la ventana de mapeo.

```
  public boolean isInsidePolygon(PVector pos) {
    int i, j =vertices.length-1;
    int sides = vertices.length;
    boolean oddNodes = false;
    for (i=0; i<sides; i++) {
        if ((vertices[i].y < pos.y && vertices[j].y >= pos.y
              || vertices[j].y < pos.y && vertices[i].y >= pos.y)
```

```
             && (vertices[i].x <= pos.x || vertices[j].x <= pos.x)) {
   oddNodes^=
            (vertices[i].x + (pos.y-vertices[i].y)/(vertices[j].y - vertices[i].y)*
            (vertices[j].x-vertices[i].x)<pos.x);
        }
        j=i;
    }
   return oddNodes;
}
```

3.5. Uso de la librería

Para el uso de la librería que se desarrolló en este proyecto, se crearon ciertos métodos que se enfocaron en la creación del `mapping` y que permiten encapsular mucho del código complejo de la librería y permitirle al usuario enfocarse en la creación de texturas y su colocación sin necesidad de involucrarse con el mapeo de textura o la corrección de la perspectiva.

```
  import proyecto.*;
 ParentView vista;
 ControllerAdministrator miControlador;

    @Override
    public void setup() {

        background(0);
        size(400, 400);
        miControlador = new ControllerAdministrator(this);

      //Definir fondo a la ventana secundaria
        miControlador.ChangeBackgroundSubview("C:\\texturas\\allien.jpg");

        //Si no se define un parametro toma la imagen por defecto de la libreria
        miControlador.ChangeBackgroundSubview();

      // miControlador.mapping.imagen2 = loadImage("imagenes/colash.jpg");
        miControlador.addContenido();
        miControlador.setup();

        //  miControlador.vista.loadImage(P2D)
    }
@Override
```

Este método debe ser llamado en el momento que se desea crear el controlador con todos sus componentes.

```
public void draw() {
        miControlador.draw();
    }
```

3.5.1. Ejemplos

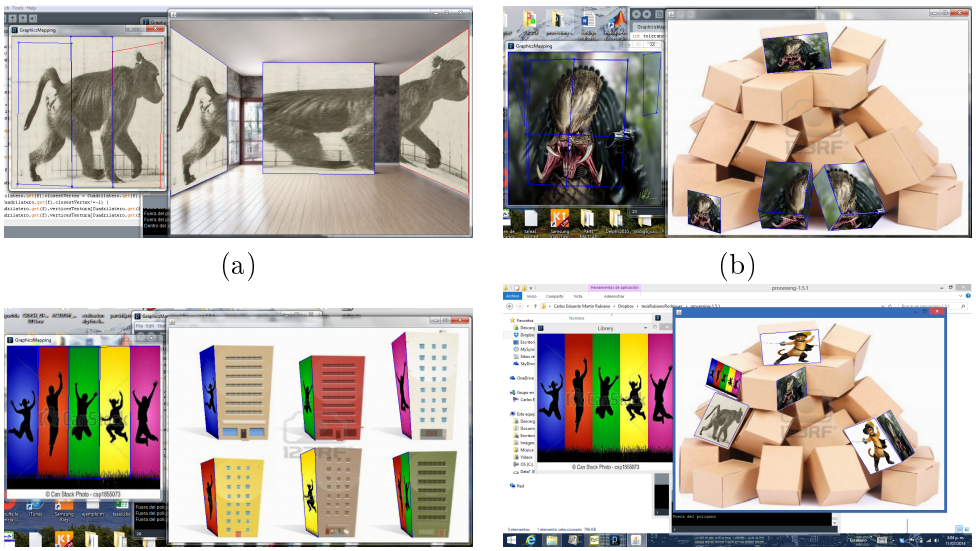
En esta Sección nos enfocaremos en mostrar algunos ejemplos del funcionamiento de la librería, A continuación se puede apreciar de varios ejemplos de como a través de distintos cuadriláteros, se puede mapear una imagen, desde la ventana principal hacia la secundaria.

Figura 3.4: ejemplo cuadrilátero puesto en escena donde los vértices se pueden deformar a gusto del usuario, con el fin de mapear determinada zona.

3.6. Conclusiones

A través de la investigación y de la creación de la librería, relacionamos diferentes conceptos de álgebra lineal, superficies paramétricas y desarrollo en Java dentro de la arquitectura de `Processing` para satisfacer las necesidades de usuarios de Processing no necesariamente ingenieros o científicos. Estas características del proyecto y el proceso de investigación nos permiten concluir que:

- El esquema de dos ventanas para el control del mapeo de las coordenadas de textura y la visión de proyección simplifica la tarea del mapping, permitiendo al usuario ver de forma inmediata y en tiempo real de la realización del mapping y proporciona una manera fácil de cargar imágenes o videos por medio del teclado.
- El contacto con los usuarios finales que conocen `Processing` pero tienen una formación diferente a ciencias e ingneiería es de gran ayuda para validar la funcionalidad y la usabilidad de una librería en el esquema de Processing. Una de las fases complicadas del desarrollo fue el mapeado, ya que al mapear una superficie irregular haciendo que nuestra proyección quede en el interior del objeto.



(a) (b)

Figura 3.5: Dos ejemplos donde los vértices se pueden deformar a gusto del usuario, con el fin de mapear determinada zona. (a) Ejemplo de varios cuadriláteros, aplicando el concepto de mapping de una misma textura sobre varios objetos (b) Ejemplo de varios cuadriláteros, aplicando el concepto de mapping de una misma textura sobre varios objetos.

- A la hora de realizar nuestro primer mapeo de proyección, somos capaces de valorar el tiempo y el trabajo que se requiere para realizar este tipo de proyecto.
- La simplicidad del código interno de la librería se da gracias a la utilización de los bloques beginShape()-endShape().
- La utilización de dos ventanas para manipular tanto las texturas a mapear como la escena a proyectar permiten al usuario un nivel de control extra sobre el mapping diferente a Quartz Composer.

3.7. Trabajo futuro

el proyecto logró cumplir con los objetivos planteados, pues se desarrolló una librería para `Processing` que permite hacer `mapping` sobre superficies planas irregulares. La librería propuesta, en la arquitectura planteada y con los requerimientos definidos permite la extensión de su funcionalidad para mejorar la interacción con el usuario final y permite la definición de esquemas de mapeo sobre superficies no necesariamente planas. En efecto, a futuro se puede implementar esquemas para rotación de los planos dentro de la ventana de proyección a través de la adición de un `Arcball` para el control del mouse[24].

Bibliografía

[1]

[2] Vistas y proyecciones.

[3] amolasmates.es. Regla de cramer.

[4] Ignacion araya y Jose tomas gonzalez. Transformaciones geométricas.

[5] ARTICS. Proyeccion.

[6] blackpawn.com. Point in triangle test.

[7] Kevin karcht Brett Jones, Raj sodhi. projection maping.

[8] F. d. f. e. purposes. Point-in-polygon algorithm — determining whether a point is inside a complex polygon, 2007.

[9] F. d. f. e. purposes. Vectores, 2012.

[10] educ.ar. El portal educativo del estado argentino, 2010.

[11] fing.edu.co. fing.edu.co.

[12] garcia. Coordenadas baricentricas.

[13] S. I. Grossman. *Algebra Lineal.* 2007.

[14] Grupo4. Tetris educativo, Enero 2013.

[15] icaro. Coordenadas homogeneas.

[16] jesus rodriguez. textura.

[17] palnoise.org. Preguntas frecuentes 3d mapping.

[18] Paulino. Computación gráfica FI: proyecciones geométricas, June 2008.

[19] processing. preocessing pgraphics.

[20] processing. preocessing textura.

[21] serdis.dis.ulpgc.e. Transformaciones geométricas.

[22] TxemaRodriguez. Un lenguaje para crear medios audiovisuales.

[23] F. C. E. (UNICEN). Análisis y diseõ de algoritmos ii y algoritmos geométricos.

[24] wikiprocessing. The arcball.

[25] H. M. y. A. Sanabria. Algebra lineal, 2008.