UNIVERSIDAD DE CONCEPCIÓN
DEPARTAMENTO DE INGENIERÍA INFORMÁTICA Y CIENCIAS DE LA COMPUTACIÓN

REPRESENTACIÓN DE LAYOUTS CURVOS MEDIANTE SPLINE PARA EL PROCESO DE PREPARACIÓN DE DATOS DE MÁSCARA DE SYNOPSYS

Trabajo de tesis para optar al grado de

Magíster en Ciencias de la Computación

POR: David Hernán Vidal González

Profesores Guía: Diego Seco Naveiras
María Andrea Rodríguez Tastets

Concepción, Chile agosto 2020

Agredecimientos

Agradezco sinceramente a todas la personas involucradas en el desarrollo de este trabajo.

Resumen

En este trabajo se presentan los resultados de la evaluación del uso de curvas de Bézier para la representación datos curvilíneos en el proceso de preparación de datos de máscara de Synopsys. Se ha propuesto una estructura de datos que permite representar los diseños, tal como lo haría una representación poligonal, pero en este caso, se propone una representación más acorde a geometrías curvas. Con base en la nueva representación, se han desarrollado las implementaciones de operaciones geométricas básicas, como las transformaciones lineales, así como también los algoritmos de clipping y healing, este último parcialmente. Habiendo hecho estas implementaciones, se desarrollaron experimentos que han permitido explorar las ventajas del uso de las curvas de Bézier, como herramienta para manejar datos curvos. De los resultados es posible desprender que la incorporación de la curvas de Bézier, resulta beneficioso desde el punto de vista de la performance, uso de memoria, y calidad de las representaciones. Sin embargo, se hace evidente de este trabajo que la implementación de algoritmos que permiten resolver las operaciones geométricas utilizando esta nueva estructura de datos, conlleva definir apropiadamente mayor número de condiciones dentro de cada algoritmo, haciendo que su implementación requiera de mayores esfuerzos.

Índice general

Índice de figuras

Índice de tablas

Nomenclatura

Acrónimos / Abreviaturas

CAD	Computing aided design
CATS	Computer aided transcription system
CI	Circuito integrado
GIS	Graphics information systems
ILT	Inverse lithography technology
MDP	Mask data preparation
OPC	ptical proximity correction
VLSI	Very large scale integrate

Capítulo 1

Introducción

El desarrollo de dispositivos semiconductores y sus aplicaciones, sin duda, han tenido un impacto innegable en nuestra sociedad, siendo parte fundamental en muchos procesos de la vida diaria. Del mismo modo, los procesos involucrados en la industria de los semiconductores se han caracterizado por su evolución constante, desde sus inicios con la creación de *layouts* a mano, hasta la actualidad, donde se hace uso de complejas herramientas de diseño automatizadas, y que junto a la evolución de las técnicas de manufactura de semiconductores, han permitido el crecimiento exponencial de esta tecnología a través de los años, consiguiendo mejoras en *performance*, consumo de potencia, área, costos, etc. [13]

Dentro de las distintas etapas del proceso de diseño de circuitos integrados (CI), en particular herramientas de preparación de datos de máscara (MDP), han tenido que abordar los nuevos requerimientos de los procesos de fabricación y de las tecnologías de la industria, tales como: fabricación de CI cada vez más pequeños, diseños basados en tecnología óptica, permitir aplicar estrategias de corrección de proximidad óptica (OPC[1]), técnicas de fotolitografía inversa (Es un enfoque riguroso para determinar las formas de máscara que producen el resultado deseado en el wafer.), etc. La incursion en estas estrategias ha implicado un auge en la necesidad de tratar *layouts* con datos curvilíneos.

Empresas dedicadas al desarrollo de software para el diseño de CI, como por ejemplo Synopsys, cuentan con herramientas que si bien permiten trabajar con datos curvilíneos, estos se basadan en una representación poligonal, haciendo para el manejo de grandes volúmenes de datos, dicha representación pueda resultar poco eficiente. Adicionalmente el uso de una representación basada en polígonos involucra la propagación de error a través de todo el flujo de operaciones geométricas a las que se somete un diseño.

Como propuesta al problema del manejo de datos curvilíneos, el grupo de CATS de Synopsys, encargado del software de MDP, propone la incorporación de una nueva repre-

[1] Correcciones del diseño debido a refracción de la luz.

sentación de los datos curvilíneos, que permita asegurar invarianza en la topología original del diseño a través del flujo de operaciones de MDP, como también que permita hacer una reducción de la cantidad de memoria necesaria para representar un *layout*. Para ello, se sugiere buscar alguna representación basada en curvas paramétricas tipo spline, las cuales a priori satisfacen los requerimientos planteados.

Este trabajo se realizará en conjunto con el grupo CATS de Synopsys Chile, del cual se espera proponer y evaluar el uso de una nueva estructura de datos que permita mejorar el manejo de datos curvos en el proceso de MDP.

Este documento esta compuesto de los siguientes capítulos: Capítulo 1: *Introducción*, es el capítulo presente. Capítulo 2: *Descripción del problema*, entrega las nociones y conexto del problema. Capítulo 3: *Estado del arte*, en él se entregan todos los contenidos teóricos para entender la soluciones actuales al problema. Capítulo 4: *Algoritmos*, se describe la estructura de datos propuesta, con la cual es posible utilizar la representación de curvas en base a curvas de Bézier, por otro lado, se detallan los algoritmos de *clippping* y *healing*, este último parcialmente, dado que una solución que permita funcionar con cualquier tipo de geometría, sale de los alcances del trabajo. Capítulo 5: *Experimentos y Resultados*, se detallan las configuraciones experimentales necesarias para estudiar el comportamiento de la *performance*, uso de memoria y calidad de la representación propuesta, en este capítulo también se entregan los resultados obtenidos. Capítulo 6: *Conclusiones*, en este se entregan las conclusiones del trabajo, asi como también los trabajos futuros.

Capítulo 2

Descripción del problema

2.1. Introducción

En este capítulo se entrega primera instancia el contexto del proceso de diseño del diseño de un CI, en las secciones siguientes se entrega una descripción a alto nivel de dicho proceso. Con ello se espera entregar los lineamientos necesarios para entender el problema y los alcances de este trabajo.

2.2. Flujo de diseño

El diseño de un CI puede ser descrito en varias etapas [7]. En dicho proceso podemos encontrar distintos niveles de representación de los diseños, en los cuales además es necesario hacer un proceso iterativo de modo de satisfacer los requerimientos impuestos en cada etapa. En la figura 2.1 se aprecian los principales pasos de este proceso.

Especificación

Es el punto de inicio de todo diseño, corresponde a una descripción de alto nivel que busca definir los objetivos, requerimientos, *performance*, disponibilidades de tecnología y dimensiones físicas de un diseño. En esta especificación también debe buscarse el balance entre los requerimientos tecnológicos y económicos involucrados.

Diseño de arquitectura

En el diseño de la arquitectura se busca decidir con qué componentes se dotará al CI con tal de satisfacer los requerimientos, se define si se usará lógica analógica, digital, o ambas,

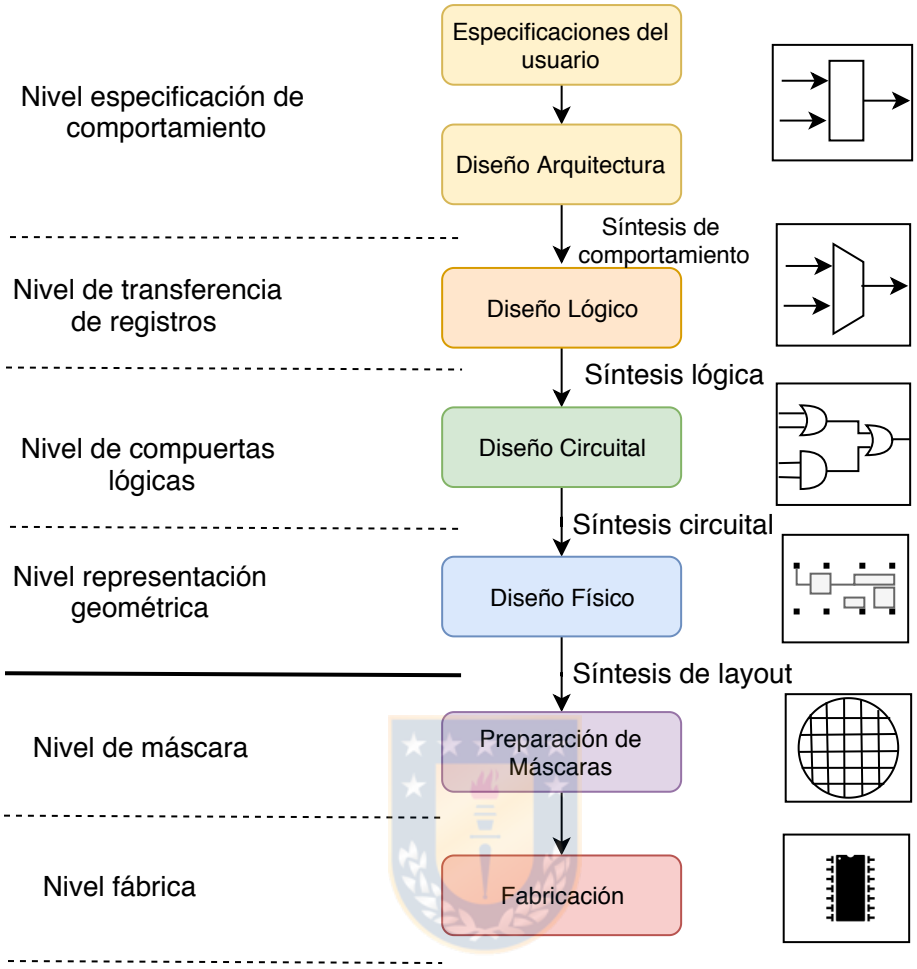


Figura 2.1 Flujo de diseño de un CI. Fuente: adaptado de [6]

se define el manejo de memoria, número de núcleos (si aplica), tipo de comunicación, uso de propiedad intelectual de terceros, y se especifican los requerimientos de potencia. En general se divide la funcionalidad del CI en varios subsistemas y se busca especificar qué arquitectura lo puede implementar.

Diseño lógico

En esta etapa se define el flujo de control de los datos, el tamaño de palabra de los registros, se hace asignación de registros y operaciones aritméticas. Se crea una representación funcional del diseño, implementado en una descripción denominada nivel de transferencia de registros (RTL), la que está expresada en algún lenguaje de descripción de hardware (HDL), tal como VHDL o Verilog.

Diseño de circuito

Una vez definida la lógica del diseño, esta es sometida a una herramienta de síntesis lógica, la cual se encarga de convertir las expresiones booleanas a una descripción optimizada a *nivel de compuertas lógicas* (netlist), es decir, una descripción expresada en componentes electrónicos y sus conexiones, para lo cual se hace uso de la descripción en RTL del paso anterior.

Diseño físico

En este paso, la representación en *netlist* se convierte a una representación geométrica. Dicha representación resultante se denomina *layout*, y se forma al convertir cada compuerta lógica, a una representación geométrica, que típicamente utiliza múltiples capas. Por otro lado, las conexiones entre los componentes también deben especificarse. El *layout* resultante está fuertemente sujeto a reglas de diseño, las cuales son una guía base sobre las limitaciones implicadas en el proceso de fabricación, del comportamiento eléctrico de los materiales debe ser considerado.

Preparación de datos de máscara

La preparación de datos de máscara (MDP) es el proceso de transformar las geometrías que conforman al *layout* del circuito integrado a un conjunto de instrucciones que pueden ser usadas para generar una fotomáscara física. En general el proceso de MDP involucra fracturar los polígonos complejos del *layout*, en figuras simples, como triángulos o trapezoides, los que resultan ser apropiados para usarlos por las impresoras de fotomáscaras.

Adicionalmente a la fractura, en el proceso de MDP, se requiere de una serie de operaciones de verificación que permitan hacer:

- Verificación de reglas de máscara, lo cual involucra:
    1. Consultas sobre espacio entre geometrías.
    2. Dimensión de una geometría.
    3. Distancia entre vértices de una geometría.
    4. Muescas.
    5. Puntos singulares.
- Verificación de proximidad óptica.
- Operaciones booleanas a las geometrías de un *layouts*.

La preparación de las máscaras requiere archivos que principalmente están escritos en estándar GDSII (Graphic Database System) [2] o más recientemente OASIS (Open Artwork System Interchange Standard). Este estándar ha tomado relevancia últimamente, ha surgido como sucesor de GDSII [4], y algunas de las mejoras que presenta son:

- Incorpora estrategias de codificación de largo variable para coordenadas, con propósitos de compresión de datos.
- Aplica algoritmos de compresión como gzip.
- Permite representar figuras más complejas que el formato GDSII, como por ejemplo, la posibilidad de describir regiones circulares.

Fabricación y empaquetado

El resultado final es almacenado en formatos GDSII, u OASIS, el que podrá entonces ser enviado a una fábrica especialista en el área. Luego de ser probada la correcta funcionalidad del CI, éste es empaquetado en algún encapsulado apropiado según el tipo de aplicación.

Así, en este trabajo se pretende abordar un problema asociado a una de las últimas etapas dentro del flujo de diseño y fabricación de los CIs, en particular se estudiarán técnicas para mejorar la representación de datos de máscara de tipo curvilíneo.

2.3. Descripción del problema

Con los nuevos requerimientos del proceso de MDP, debido al auge de datos curvos, es nesario mejorar la representación poligonal, debido a los siguientes problemas:

1. Requiere incorporar muchos puntos para representar datos curvos.
2. En las aproximaciones hechas para representar datos curvos, no se preserva la topología original del diseño ante ciertas operaciones.

Ante estos problemas es que se busca pensar en una representación basada en un modelo matemático más acorde a las curvas.

2.4. Objetivos

Se busca estudiar la incorporación al proceso de MDP de una representación que haga uso de curvas de Bézier, para el manejo de datos curvos.

Se propone una estructura de datos que permite por una lado mantener las propiedades de la representaciones polígonos, pero que también permite hacer uso de curvas de Bézier para el manejo de datos curvos.

Junto con ello se deberá implementar un conjunto de operaciones geométricas básicas como transformaciones lineales, además de los algoritmos de clipping y healing.

Para cada implementación propuesta se deberán comparar, respecto a una representación poligonal equivalente, en las métricas de *performance*, uso de memoria y calidad de las representaciones. Esto con el propósito de cuidar que ninguna de ellas se salga del orden de magnitud de la representación de comparación base.

Se deberá considerar que los datos de entrada son artificiales, los que serán provistos para los experimentos mediante la estructura de datos propuesta.

Capítulo 3

Estado del arte

3.1. Introducción

En este capítulo se pretende entregar los fundamentos a cada una de las soluciones existentes actualmente, en particular se trata de las definiciones de las transformaciones lineales, aplicadas a polígonos, las cuales pueden ser extendidas fácilmente a curvas spline. Por otro lado, se entregan las descripciones de los algoritmos de *clipping* y *healing*, también para una representación poligonal, las cuales se usaron como base para definir los algoritmos para spline propuestos en este trabajo. Finalmente se entrega la definición de las curvas de Bézier, las cuales en la práctica consituyen la elección de curvas paramétricas usadas en este trabajo.

3.2. Transformaciones

3.2.1. Introducción

Las transformaciones juegan un importante rol en computación gráfica. Estas permiten producir modificaciones en una representación.

3.2.2. Transformaciones Lineales

Definición 3.2.1. Un n-espacio euclideano es $\mathbb{R}^n = \{x = (x_1, x_2, ..., x_n)\}$ donde x es un punto de n coordenadas.

Definición 3.2.2. Una transformación en $\mathbb{R}^n$, es una relación $T : \mathbb{R}^n \longrightarrow \mathbb{R}^n$, tal que, para cada punto $X \in \mathbb{R}^n$ se relaciona con un único punto $T(X) \in \mathbb{R}^n$.

Definición 3.2.3. Sea $T : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ una transformación. Se dice que T es una transformación lineal si y solo si:

1. $\forall \alpha \in \mathbb{R}$ y $\forall X \in \mathbb{R}^n$ se cumple que: $T(\alpha x) = \alpha T(X)$

2. $\forall X, Y \in \mathbb{R}^n$ se tiene que: $T(X+Y) = T(X) + T(Y)$

3.2.3. Traslación

Definición 3.2.4. Traslación
Sea $T : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ una transformación. T se dice una traslación si existe $Y \in \mathbb{R}^n$ tal que $\forall X \in \mathbb{R}^n$ se tiene que $T(X) = X + Y$. Es decir una traslación mueve todos los puntos en una determinada distancia y dirección.

Ejemplo 3.2.1. *Considerar la transformación $T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$, definida como:*

$$T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2 : \begin{bmatrix} P'_{ix} \\ P'_{iy} \end{bmatrix} = \begin{bmatrix} P_{ix} \\ P_{iy} \end{bmatrix} + \begin{bmatrix} \triangle x \\ \triangle y \end{bmatrix}$$

En la figura 3.1, la transformación lineal de traslación T ha sido aplicada sobre todos los puntos P_i del polígono P, desplazando los puntos de P las cantidades $\triangle x$ y $\triangle y$, en las direcciones del eje x e y respectivamente.

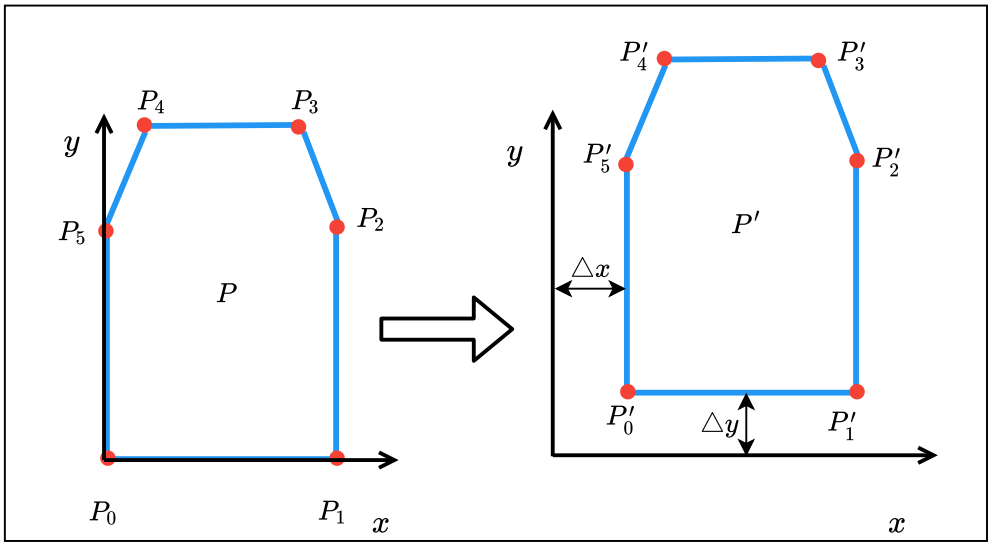


Figura 3.1 Transformación de traslación. Fuente: elaboración propia.

3.2.4. Escalado

Definición 3.2.5. Escalado uniforme
Sea $T : \mathbb{R}^n \longrightarrow \mathbb{R}^n$ una transformación. T es una transformación de escalado uniforme, si $\forall X \in \mathbb{R}^n$ y $\alpha \in \mathbb{R}$ se tiene que $T(X) = \alpha X$

Ejemplo 3.2.2. *Considerar la transformación $T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2$, definida como:*

$$T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2 : \begin{bmatrix} P'_{ix} \\ P'_{iy} \end{bmatrix} = \begin{bmatrix} \alpha P_{ix} \\ \alpha P_{iy} \end{bmatrix}$$

En la figura 3.2 la operación de escalado T ha sido aplicada sobre todos los puntos P_i del polígono P.

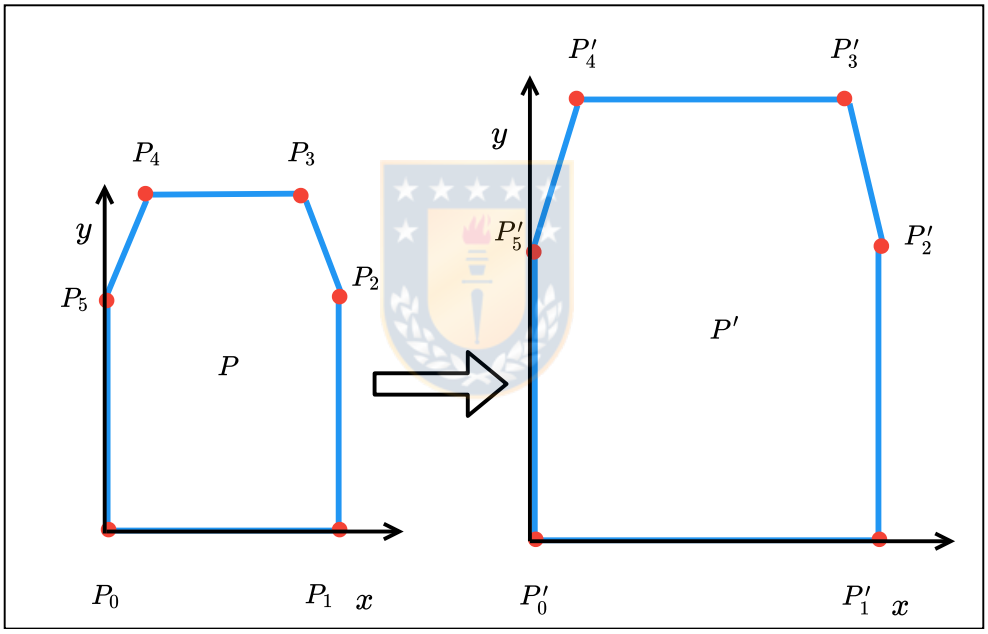


Figura 3.2 Transformación de escalado. Fuente: elaboración propia.

3.2.5. Rotación

Definición 3.2.6. Rotación en $\mathbb{R}^2$
Sea $T : \mathbb{R}^2 \longrightarrow R^2$ una transformación. T se define como la transformación de rotación del plano, con ángulo θ medido desde el *eje x* contra reloj como:

$$T : \mathbb{R}^2 \longrightarrow \mathbb{R}^2 : \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Ejemplo 3.2.3. *La transformación de rotación T de la definición anterior, es aplicada sobre todos los puntos P_i del polígono P.*

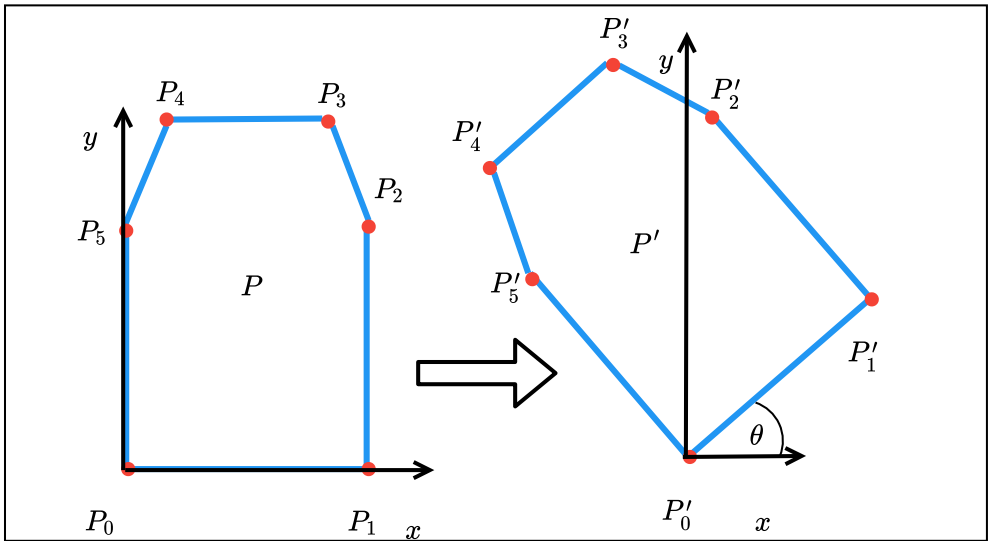


Figura 3.3 Transformación de rotación *(Fuente: elaboración propia)*.

Así, cada una de las definiciones de las transformaciones lineales mostradas, resultan fácilmente extensibles a una representación basada en curvas.

3.3. Clipping

3.3.1. Introducción

Clipping es una de las operaciones fundamentales en geometría computacional, y presenta aplicaciones en las áreas de computación gr'afica y diseño asistido por computador (CAD) [9]. Algunos de sus usos se encuentran en diseño de componentes con integración a muy gran escala (VLSI), superposición de mapas en sistemas de información grafica (GIS) [10] , distribución de tareas para el procesamiento distribuido [5], entre otras.

En esta sección se entrega una descripción básica de en qué consiste *clipping*, luego se mencionan al menos 4 algoritmos que resuelven el problema de *clipping* en polígonos, de entre los cuales se explican más detalladamente los algoritmos de Sutherland-Hodgman y Weiler–Atherton, dado que en ellos se basa el algoritmo de *clipping* propuesto en este trabajo.

3.3.2. Descripción básica de *clipping*

El problema de *clipping* consiste en que dadas 2 geometrías, una denominada agente y otra sujeto, determina cuál es la porción del sujeto que está contenida dentro del agente. De manera equivalente, esta operación encuentra la intersección entre agente y sujeto, tal como se muestra en la figura 3.4.

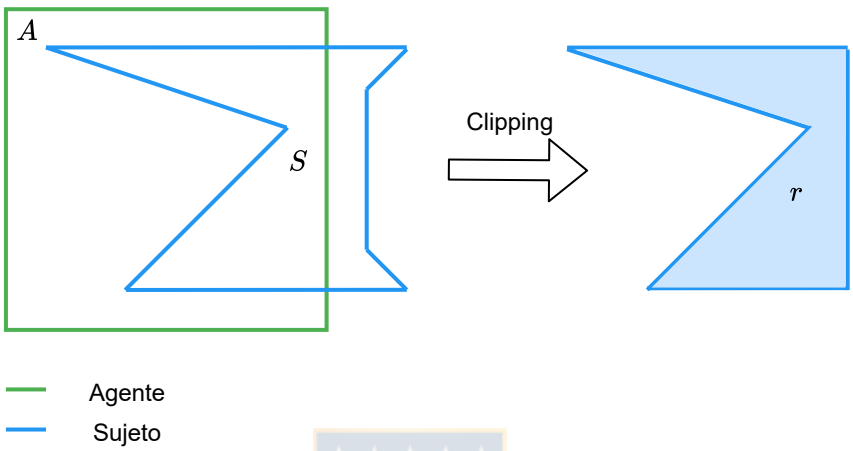


Figura 3.4 Operación de *clipping* entre el agente A y el sujeto S, produce la geometría resultante r *(Fuente: elaboración propia)*.

3.3.3. Algoritmos de *clipping* sobre polígonos

En la literatura existe una amplia lista de algoritmos que permiten determinar *clipping* sobre polígonos, algunos de ellos son:

- Sutherland-Hodgman, algoritmo que se caracteriza por ser simple de implementar, pero está limitado a trabajar con polígonos convexos, en otro caso, puede producir geometrías degeneradas, como aristas colineales [11].
- Weiler–Atherton, algoritmo de *clipping* general, es decir, puede trabajar con polígonos convexos y cóncavos, los que incluso pueden contener hoyos [3].
- Vatti: algoritmo de *clipping* general, es más eficiente que los anteriores, está basado en el paradigma de sweep line [8], pero es más difícil de implementar. Puede ser extendido para determinar algunas operaciones booleanas, como unión o diferencia [12].
- Greiner–Hormann: algoritmo más eficiente que el algoritmo de Vatti. Está basado en el cálculo del número de *winding*. Al igual que el algoritmo anterior, puede ser extendido para realizar otras operaciones booleanas [5].

3.4. Sutherland-Hodgeman

3.4.1. Introducción

El algoritmo de Sutherland-Hodgman se aplica para determinar el *clipping* de polígonos convexos, y puede ser extendido al espacio tridimensional para trabajar con poliedros convexos.

3.4.2. Descripción básica

Sutherland-Hodgman recibe como agente a un polígono convexo, y como sujeto un polígono no necesariamente convexo. En su salida produce un polígono equivalente a la intersección entre agente y sujeto, sin embargo, puede producir ciertas degeneraciones de salida [11].

Este algoritmo funciona de la siguiente manera, se itera sobre cada una de las aristas del agente, las cuales se extienden infinitamente, de modo de ir recortando las partes del sujeto que estén dentro de un semiplano, delimitado por la recta infinita. El proceso se repite por cada arista del sujeto.

Así, eligiendo la arista derecha del agente de la figura 3.5, el algoritmo procesa las aristas del sujeto de la siguiente forma:

1. Arista $\overrightarrow{P_0P_1}$: los puntos P_0 y P_1, son agregados a la lista de salida, dado que están contenidos dentro del semiplano de interés, figura 3.5-a.
2. Arista $\overrightarrow{P_1P_2}$: se calcula el punto de intersección P_1', y se agrega a la lista de salida, figura 3.5-b.
3. Arista $\overrightarrow{P_2P_3}$: P_2 está completamente fuera del semiplano de interés, no se almacena, figura 3.5-c.
4. Arista $\overrightarrow{P_3P_0}$: se calcula el punto de intersección P_2', y se agrega a la lista de salida, figura 3.5-d.

Este procedimiento es descrito en el Algoritmo 1.

3.4.3. Características

Una característica importante del algoritmo de Sutherland-Hodgman es que se pueden producir degeneraciones. Estas se producen cuando el plano de *clipping* produce dos o más regiones disjuntas, produciendo que estas queden conectadas por aristas colineales. Por

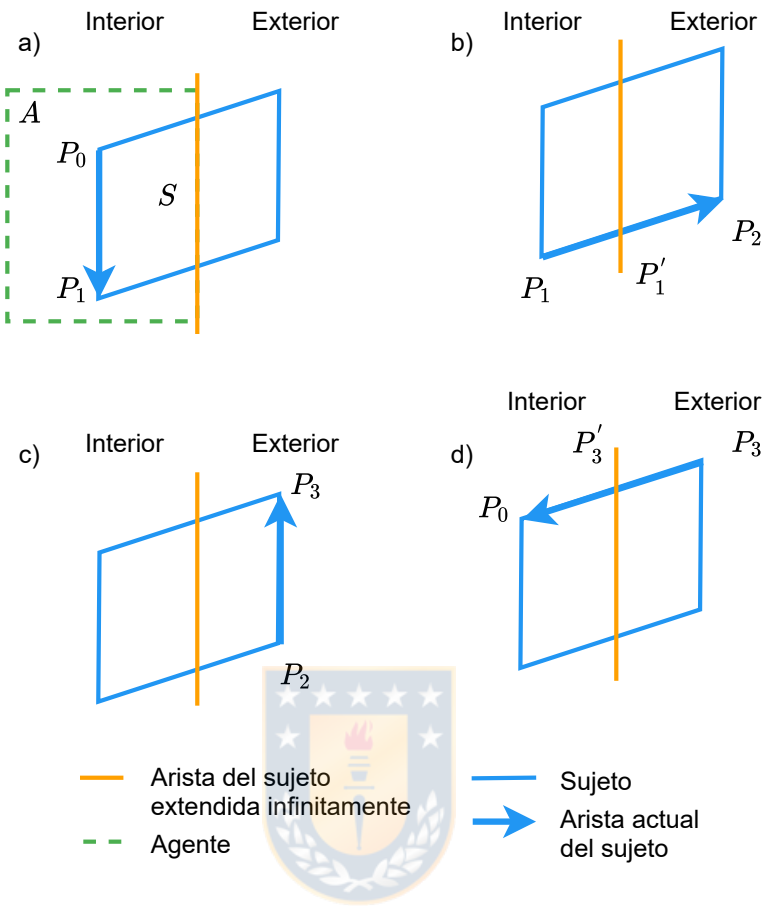


Figura 3.5 Algoritmo de Sutherland-Hodgman, se elige la arista derecha del agente, luego se procesan las aristas del sujeto, figuras de a) a d). Fuente: elaboración propia.

ejemplo, en a) de la figura 3.6, al hacer el *clipping* entre el agente A y el sujeto S, produce una geometría de salida r, sin embargo, en el caso b), esta operación produce como salida dos regiones r_1 y r_2 unidas por aristas colineales.

Algoritmo 1: Sutherland-Hodgman

Input: (SubjectGeometery, AgentGeometry)
Output: ClippedGeomtry

```
1  for semiPlaneClipEdge in AgentGeometry do
2      for CurrentPoint in SubjectGeometery do
3          PrevPoint = SubjectGeometery[CurrentPoint.prev()]
4          IntersectionPoint = ComputeIntersection(PrevPoint, CurrentPoint, agentEdge
           )
5          if CurrentPoint inside semiPlaneClipEdge then
6              if PrevPoint not inside semiPlaneClipEdge then
7                  outputList.add(IntersectionPoint))
8              outputList.add(CurrentPoint))
9          else if PrevPoint inside semiPlaneClipEdge then
10             outputList.add(IntersectionPoint)
```

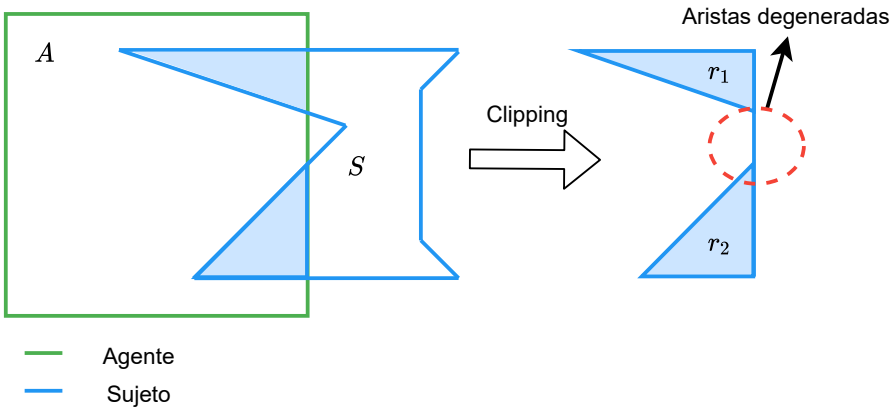


Figura 3.6 Algoritmo de Sutherland-Hodgman, en este caso se produce un par de aristas degeneradas *(Fuente: elaboración propia)*.

3.5. Weiler-Atherton

3.5.1. Introducción

Al igual que Sutherland-Hodgman, el algoritmo de Weiler-Atherton permite determinar la operación de *clipping* de una geometría, sin embargo, es posible aplicarlo a geometrías convexas, sin que se produzcan casos degenerados, como por ejemplo, la generación de aristas colineales.

3.5.2. Descripción básica

La idea principal de Weiler-Atherton consiste no solo en recorrer la geometría que actúa como sujeto, sino también partes de la geometría que actúa de agente, y lo hace según las reglas siguientes:

- **Regla 1**: si al recorrer los vértices del sujeto se pasa desde afuera hacia adentro del agente, entonces continuar almacenado los vértices del sujeto.
- **Regla 2**: si al recorrer el sujeto, se sale hacia afuera del agente, entonces recorrer este último hasta encontrar otra intersección de tipo entrante.

El algoritmo de Weiler-Atherton aplicado sobre las geometrías de las figura 3.7, se explica en los pasos siguientes:

1. Considerar como precondición que los polígonos de entrada poseen una misma orientación.
2. Hacer una lista de vértices que representen al agente y al sujeto, que incluyan los puntos de intersección entre ellos, figura 3.7.
3. Etiquetar los puntos de intersección como entrante o saliente.
4. Comenzar buscando la primera intersección entrante de la lista del sujeto.
5. Poner en la lista de salida todos los puntos de la lista del sujeto, ubicados entre una intersección entrante y una saliente (**Regla 1**).
6. Luego, cambiar a la lista del sujeto, y buscar la intersección anterior, figura 3.8.
7. Recorrer la lista del agente hasta encontrar una intersección entrante (**Regla 2**).
8. Ahora la lista de salida contendrá la primera geometría, resultante, figura 3.9.
9. Repetir el proceso hasta que todas las intersecciones entrantes sean visitadas.

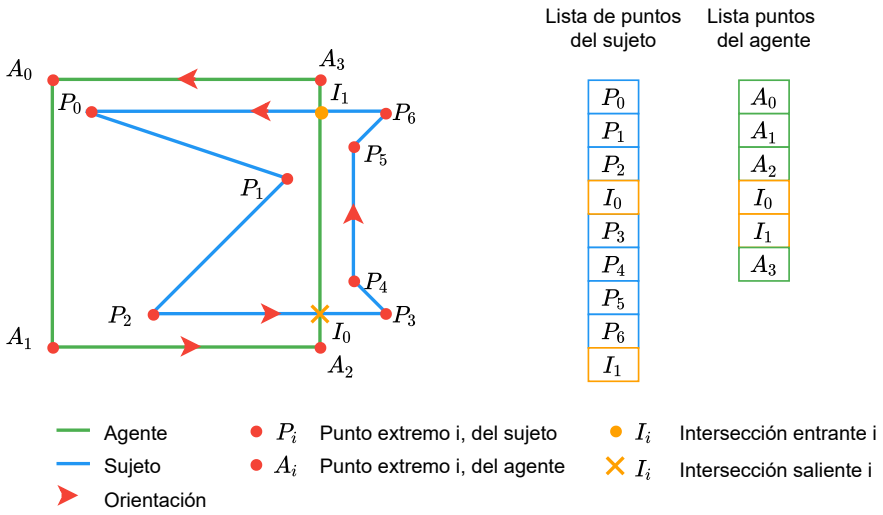


Figura 3.7 Algoritmo de Weiler-Atherton, precondición geometrías con la misma orientación, lista de vértices del agente y sujeto con sus puntos de intersección *(Fuente: elaboración propia)*.

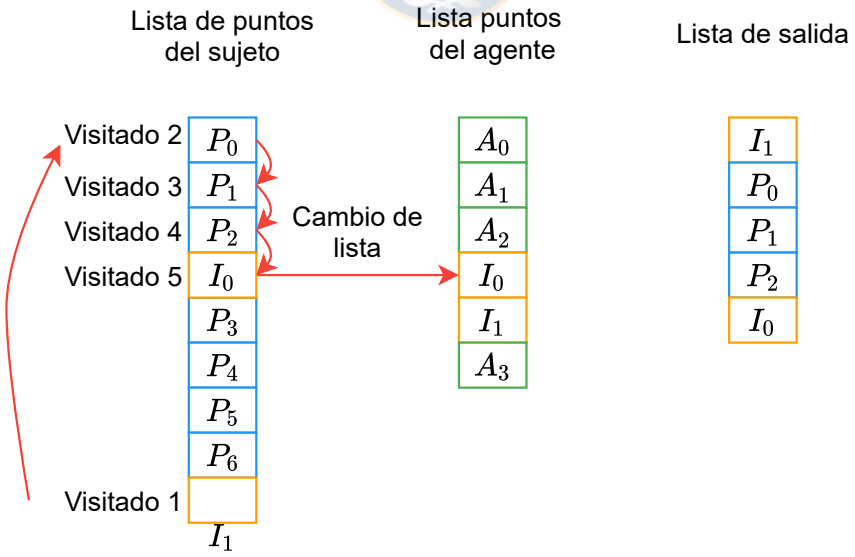


Figura 3.8 Algoritmo de Weiler-Atherton: pone en la lista de salida los puntos de la lista del sujeto que están entre una intersección entrante y una saliente, luego cambia a la lista del agente *(Fuente: elaboración propia)*.

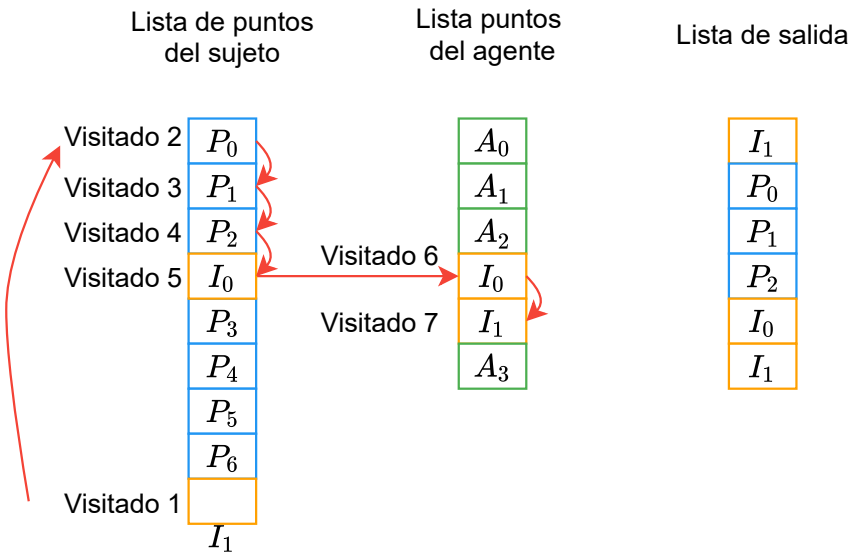


Figura 3.9 Algoritmo de Weiler-Atherton: finalmente se recorre la lista de puntos del agente hasta que encuentra una intersección entrante *(Fuente: elaboración propia)*.

3.6. Healing

3.6.1. Introducción

Healing también es considerada como una de las operaciones booleanas fundamentales en geometría computacional. En esta sección se presenta una de las versiones del algoritmo de *healing* implementadas en CATS, basada en polígonos. Este algoritmo puede ser extendido a una versión sobre una representación basada en splines, habiendo abordado apropiadamente los problemas de intersección segmento/curva e intersección curva/curva.

3.6.2. Descripción básica de *healing*

La idea de *healing* es que dadas 2 o más geometrías, retorna una geometría equivalente a la suma o unión de todas ellas. Como en figura 3.10, *healing* entre las geometrías C_0, C_1 y C_2 retorna la geometría C.

3.6.3. *Healing* basado en polígonos

Una de las versiones del algoritmo de *healing* basado en polígonos, y que es usado en CATS actualmente, posee los siguientes pasos:

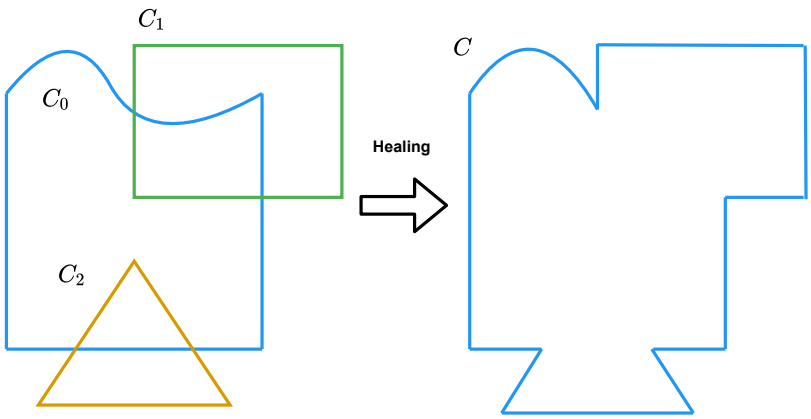


Figura 3.10 *Healing* de las geometrías c_0, c_1 y c_2 *(Fuente: elaboración propia).*

1. **Entrada**: como entrada este algoritmo recibe un conjunto de segmentos que representan las geometrías a operar. Los segmentos no necesitan seguir la continuidad de la geometrías que describen, es decir, cada segmento es tratado como un elemento independiente.

2. **Resolver intersecciones**: se requiere encontrar todas las intersecciones de tipo segmento/segmento, y subdividir aquellos segmentos en sus puntos de intersección, de modo de que los segmentos resultantes a lo sumo se intersecten en sus puntos extremos. Como precondición del paso siguiente, se deben ordenar los segmentos desde abajo hacia arriba, y de izquierda a derecha.

3. **Generar clústers**: se refiere a agrupar los segmentos anteriores, de forma tal que dentro de un clúster, se agrupen todos los segmentos que tengan un punto extremo en común. Notar que un segmento puede pertenecer a 1 ó 2 clústers.

4. **Procesar clústers**, en este paso se pretende eliminar todos aquellos segmentos pertenecientes a un clúster que no serán parte de la geometría resultante, esto se hace valiéndose de una lista activa de segmentos, y de una regla booleana, **or** en el caso de healing y **and** para intersección. En este paso la aplicación de una regla booleana en particular permite realizar operaciones booleanas distintas.

5. **Generar contornos**, finalmente una vez seleccionados los segmentos útiles dentro de un clústers, estos deben unirse con tal de generar la geometría resultante.

Resolver intersecciones

El primer paso para este algoritmo es resolver las intersecciones de tipo segmento/segmento. Para ello, se deben definir los tipos de intersección, y basado en el número de segmentos que retornan, se encuentran:

- Segmento/segmento:
  1. 0 segmentos de retorno, figura 3.11-a.
  2. 2 segmentos de retorno, figura 3.11-b.
  3. 3 segmentos de retorno, figura 3.11-c y figura 3.11-e.
  4. 4 segmentos de retorno, figura 3.11-d.

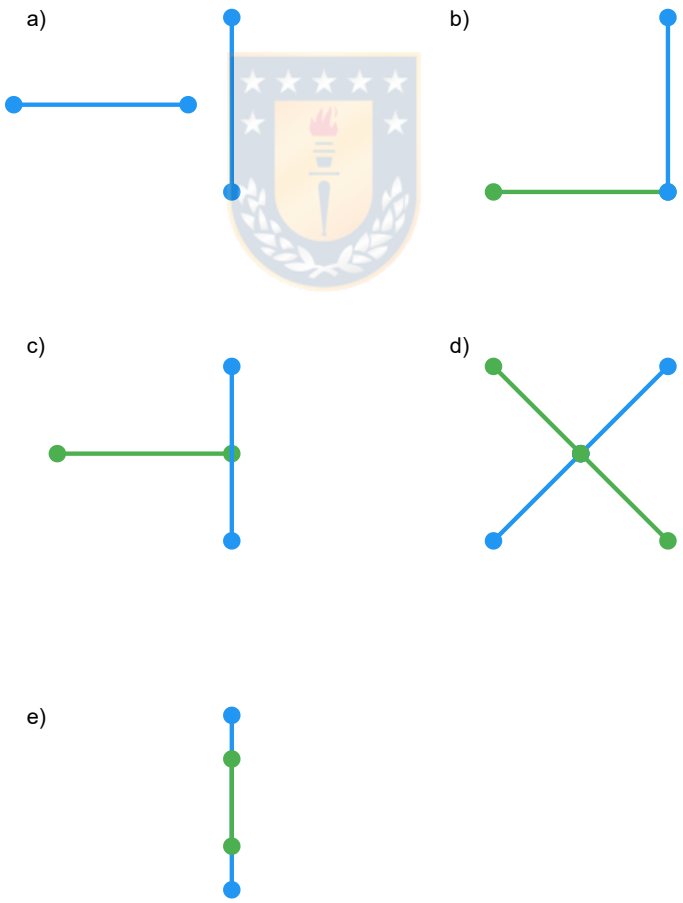


Figura 3.11 Eventos de intersección segmento/segmento *(Fuente: elaboración propia)*.

Generar Clústers

Uno de los elementos importantes dentro del algoritmo propuesto de *healing* son los clústers. Un clúster almacena a todos los segmentos que comparten un punto extremo en común, y lo hace con 2 listas, una de ellas almacena a los segmentos que están por sobre la horizontal que cruza al punto común del clúster, y otra lista guarda los segmentos que se ubican por debajo de dicha horizontal. Los segmentos asociados a estas listas se denominan de *leading* y *trailing*, respectivamente. Adicionalmente, cada lista debe mantener sus segmentos con un orden específico, esto es, en la lista de leading, los segmentos se ordenan en forma horaria, mientras que en la lista de trailing se ordenan en forma antihoraria.

Notar que aquellos clústers ubicados en la zona más baja de la geometría tendrán una lista de *leading* vacía, mientras que los clústers ubicados en la parte superior de la geometría, tendrán una lista de *trailing* vacía.

Adicionalmente los clústers deben satisfacer un criterio de orden, dado por el algoritmo 2.

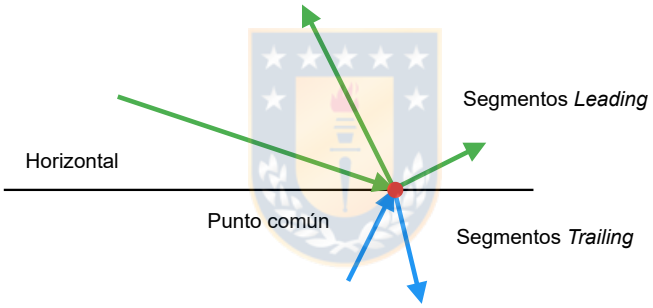


Figura 3.12 Elementos de un clúster, punto común, segmentos de *leading* y de *trailing* *(Fuente: elaboración propia).*

Procesar clústers

Esta etapa se encarga se seleccionar los segmentos pertenecientes a un clúster que aportan datos a la geometría de salida. La estrategia para ello está basada en sweepline, la que se mueve entre los puntos comunes de un cluster, y cuya lista activa se encarga de 2 cosas:

1. Almacenar los segmentos en forma ordenada, que han aparecido en las lista de leading, y que no hayan aparecido como segmentos de trailing.

2. Por cada segmento, se debe calcular un contador de figura, respecto a todos los segmentos que se encuentran a su izquierda en la lista activa. El contafor de figura se

basa en que cada segmento (que no esté describiendo un hollo de la geometría) de un *layout*, posee información gométrica a su izquierda.

El procedimiento para llenar la lista activa es el siguiente:

a) Recorrer los clústers desde la más abajo hacia arriba.

b) Insertar el primer segmento de la lista de leading del primer clúster.

c) Calcular el número de winding(respecto de los elementos a su izquierda) del segmento.

d) Insertar en forma ordenada (de izquierda a derecha) los siguientes segmentos de leading del cluster, e ir calculando los números de winding.

e) Segmentos útiles son aquellos en los que se producen las siguientes transiciones entre los números de winding.

f) 1) Si la transición del número de winding va desde 0 a un número mayor que 0, entonces es un segmento que sirve. Implicitamente en este punto se está aplicando la regla booleana **or**.

2) Si la transición del contador de figura de un segmento va desde un número mayor que cero a cero, también se considera como un segmento útil. Del mismo modo que antes, implicitamente en este punto se está aplicando la regla booleana **or**.

g) Luego pasar el siguiente clúster, quitar de la lista activa los segmentos de trailing, y repetir el procedimiento anterior.

Para garantizar el invariante de esta parte del algoritmo, se requieren 2 criterios de orden, el primero consiste en mantener ordenados los clústers, desde izquierda a derecha y desde abajo hacia arriba.

Además, la lista activa requiere definir un criterio de orden para organizar los elementos, con tal dar consistencia al contador de figura, esto es, por cada elemento si se proyecta su horizontal a la izquierda, su contador de figura coincida con el número de winding.

Generar contornos

El paso final cosiste en unir los segmentos útiles de cada clúster, para ello se deben unir los puntos coincidentes de cada cluster, hasta que se detecta que la geometría se cierra.

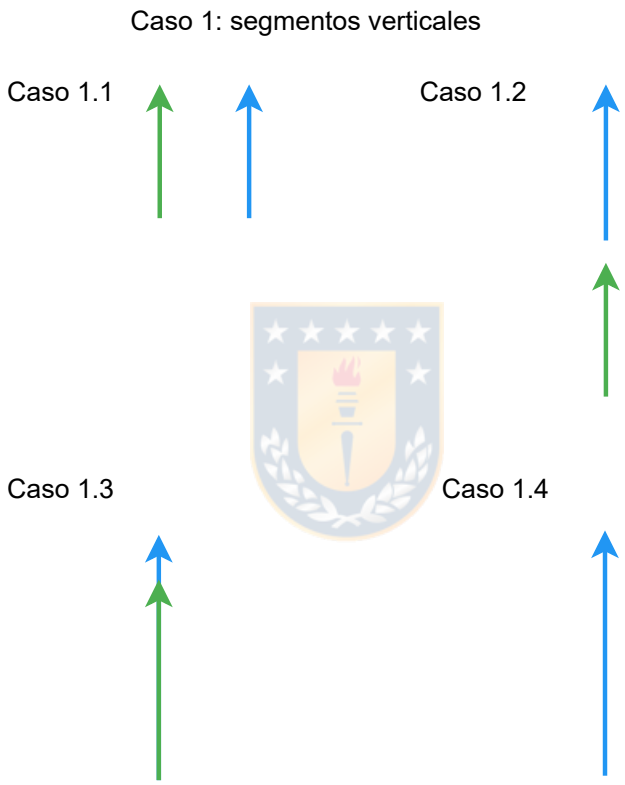


Figura 3.13 Eventos de intersección segmento/segmento *(Fuente: elaboración propia)*.

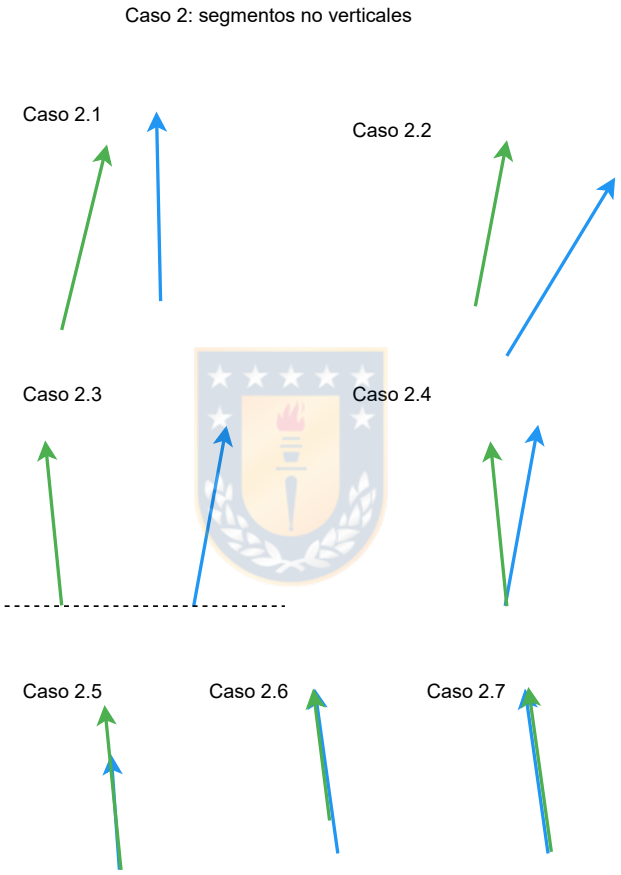


Figura 3.14 Eventos de intersección segmento/segmento *(Fuente: elaboración propia)*.

Algoritmo 2: Active list comparator

```
   Input: SegmentOrCurve left, SegmentOrCurve right
1  Point bottomLeft = left.bottom()
2  Point bottomRight = right.bottom()
3  Point topLeft = left.top()
4  Point topRight = right.top()
5  Coordinate buttomLeftX = bottomLeft.x
6  Coordinate buttomRightX = bottomRight.x
7  Coordinate buttomLeftY = bottomLeft.y
8  Coordinate buttomRightY = bottomRight.y
   /* Horizontal segment cases                                          */
9  if left.isVertical() and right.isVertical() then
10     if buttomLeftX != buttomLeftX then
11         return buttomLeftX < buttomLeftX
12     if buttomLeftY != buttomRightY then
13         return buttomLeftY < buttomRightY
14     Coordinate topLeftY = topLeft.y
15     Coordinate topRightY = topRight.y
16     if topLeftY != topRightY then
17         return topLeftY < topRightY
18     return false
   /* Non horizontal segments                                           */
19 if buttomLeftY < buttomRightY then
20     sign = CrossProduct(bottomRight - bottomLeft, topLeft - bottomLeft); if sign != 0 then
21         return sign > 0
22 else
23     buttomRightY < buttomLeftY
24 sign = CrossProduct(bottomLeft - bottomRight, topRight - bottomRight); if sign != 0 then
25     return sign < 0
26 else
27     buttomLeftY == buttomRightY
28 if buttomLeftX != buttomLeftX then
29     return buttomLeftX < buttomLeftX
30 sign = CrossProduct(topRight - bottomRight, topLeft - bottomLeft)
31 if sign != 0 then
32     return sign > 0
33 if topLeft != topRight then
34     return topLeft.y < topRight.y or (topLeft.y == topRight.y and topLeft.x < topRight.x)
35 if bottomLeft != bottomRight then
36     return bottomLeft.y < bottomRight.y or (bottomLeft.y == bottomRight.y and bottomLeft.x < bottomRight.x)
37 return false
```

Ejemplo

Los clústers para el ejemplo de la figura 3.15, con sus listas de leading y trailing son:

- c_1, L : $\{s_9, s_0\}$, T: $\{\}$
- c_2, L : $\{s_1\}$, T: $\{s_0\}$
- c_3, L : $\{s_8\}$, T: $\{s_9\}$
- c_4, L : $\{s_7, s_6, s_2\}$, T: $\{s_1\}$
- c_5, L : $\{s_3\}$, T: $\{s_2\}$
- c_6, L: $\{s_4, s_5\}$, T: $\{s_8, s_7\}$
- c_7, L: $\{\}$, T: $\{s_5, s_6\}$
- c_8, L: $\{s_3\}$, T: $\{s_4\}$

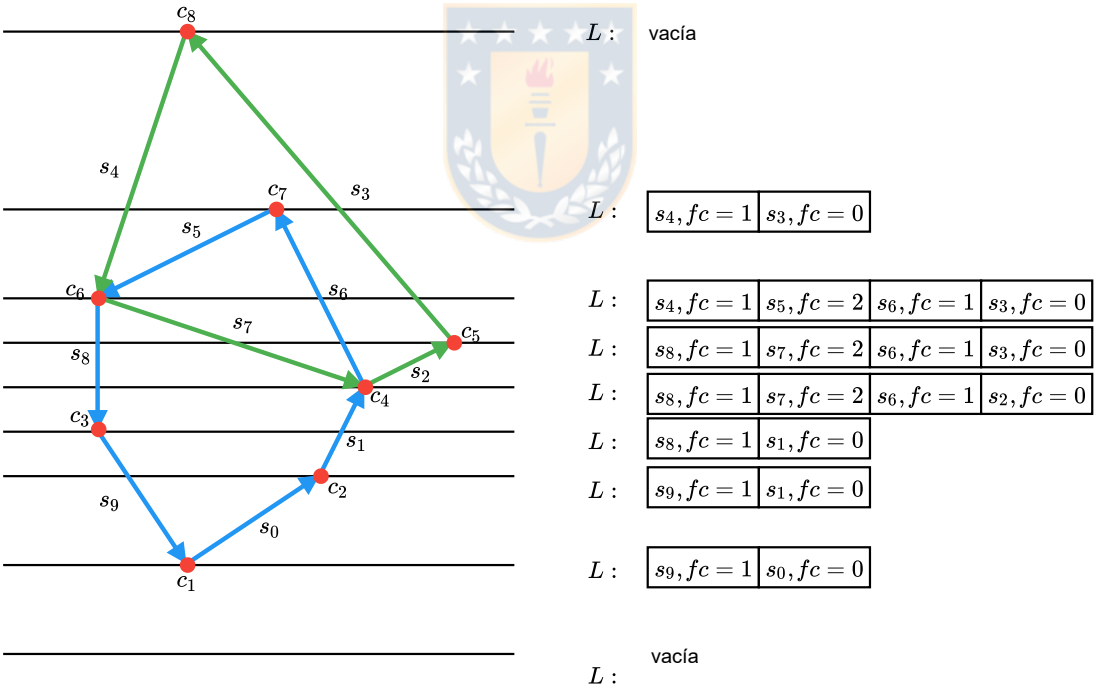


Figura 3.15 Lista activa de los segmentos del polígono. Fuente: elaboración propia.

Por otro lado, los segmentos útiles, detectados mediante la lista activa de la figura 3.15 de cada clúster son:

- c_1, $\{s_9, s_0\}$

- c_2, $\{s_1\}$
- c_3, $\{s_8\}$
- c_4, $\{s_2\}$
- c_5, $\{s_3\}$
- c_6, $\{s_4\}$
- c_7, $\{\}$
- c_8, $\{s_3\}$

En el apéndice A.2, se muestra paso a paso cómo evoluciona la lista activa. Por otra parte en el apéndice A.3, se grafican los segmentos útiles de cada clústers. Finalmente en el apéndice A.4, se indica cómo se construye la geometría resultante.

3.7. Curvas de Bézier

Durante la década de 1940 los diseñadores de avión usaban pequeñas tiras de madera, denominadas splines, para delinear las curvaturas de aeronaves. Dichas splines eran fijadas en su posición mediante pesos.

Definiciones formales de estas curvas se deben a Pierre Bézier y Paul de Casteljau alrededor de 1960, quienes le dieron aplicaciones en la industria automotriz. Hoy en día consitituyen a una de las más usadas herramientdas matemáticas para la representacions de curvas y superficies en el diseño asistido por computador.

Las curvas de Bézier están basadas en polinomios, poseen un conjunto de propiedades que las hacen apropiadas para muchas aplicaciones. Una curva de Bézier de grado n posee un conjunto $n+1$ *puntos de control*, los que al ser unidos forman un *polígono de control*.

3.7.1. Curvas de Bézier lineales

Una curva de Bézier lineal se define, usando dos puntos de control $P_0 = (x_0, y_0)$ y $P_1 = (x_1, y_1)$, por:

$$B(t) = (1-t)P_0 + tP_1$$

con $t \in [0,1]$

3.7.2. Curvas de Bézier cuadráticas

Una curva de Bézier cuadrática se define, usando tres puntos de control $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$, por:

$$B(t) = (1-t)P_0 + 2(1-t)P_1 + tP_3$$

con $t \in [0,1]$

Notar que el triángulo formado por los puntos P_0, P_1 y P_2 es denominado el polígono de control.

3.7.3. Curvas de Bézier generales

Curva de Bézier. Sea $(P_i = (x_i, y_i)\, i = 0,1,2,...,n)$ los puntos de control de una CB. Entonces una CB de grado n queda definida como:

$$P(t) = \sum_{i=0}^{n} B_i^n(t) P_i \qquad 0 < t < 1 \tag{3.1}$$

Donde $B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-1}$, con $i = 0,1,2,...,n$ son polinomios de Bernstein.

Las propiedades de una CB son las siguientes:

- Transformaciones afines: una CB es invariante ante transformaciones afines, esto quiere decir que su forma no se modifica ante traslaciones ni rotaciones.
- Transformaciones de parámetros afines: las CBs son invariantes bajo transformaciones de parámetros afines, esto quiere decir que si una curva está definida en el intervalo $[0,1]$, al aplicarle una transformación que lleve los valores de $[0,1]$ a uno $[a,b]$ con $a \neq b$, entonces la curva mantiene sus mismas propiedades.
- Propiedad de Convex-Hull: una CB está contenida en el convex hull de sus puntos de control.
- Puntos extremos: los puntos de control iniciales y finales de una CB son puntos contenidos en la curva.
- Control local: a excepción de los puntos de control inicial y final, cualquier modificación de un punto de control interior de una CB, modifica a todos los puntos de la curva.

- Seguimiento de los puntos de control: intuitivamente una CB siempre seguirá la forma del polígono formado por los puntos de control.
- Algoritmo de subdivisión: existe un método denominado algoritmo de Casteljau [1], con el que es posible obtener dos subdivisiones de una CB, además de ser útil para el cálculo de sus derivadas, entre otras aplicaciones.

Capítulo 4

Algoritmos

4.1. Introducción

Teniendo en cuenta las soluciones dadas actualemnte al problema de representar datos geométricos, con una estratégia basada en una representación poligonal, en este capítulo corresponde extender dichas funionalidades, a una representación que permite utilizar curvas de Bézier para dicho propósito. Por tanto, en este se inica expliando la estrutura d edatos que permite el manejo de curvas. Adicionalmente se muestran los algoritmos en su base

4.2. Estructura de datos propuesta

El argunmento par introducir la represnetación de datos curvos mediante curvas de Bézier, se puede entender fácilmente mediante las figuras 4.1-a y 4.1-b, en la primera de ellas representa la solución actual almenejo de datos curvos, la cual s ebasa en aproximar mediante segmentos una porcion curva.

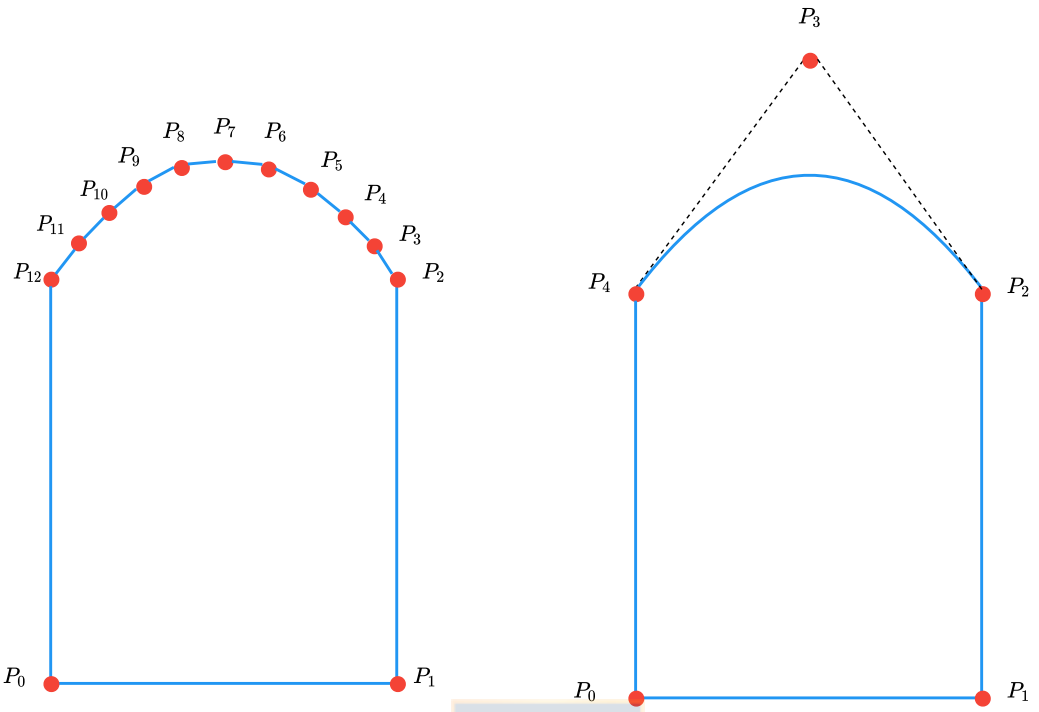
Por otro lado, en 4.1-b resulta evidente que al introducir una curva de Bézier, se puede represnetar una proción completa del diseño, pero con un número de puntos menor.

Los requerimientos para representar datos curvilíneos, son los siguientes:

a) Ser capaz de mantener las propiedades de la representación poligonal.

b) Permitir la representación de datos curvos mediante curvas de bézier.

Así, la estructura de datos propuesta está compuesta de dos vectores, uno de ellos se denomina vector de puntos de control, en el que se lamecena un cunjunto



(a) Representación poligonal de una geomtría con una sección curva representada por segmentos *(Fuente: elaboración propia).*

(b) Representación de una geomtría con un apocrcion curva represnetada por una curva de Bézier de grado 2. *(Fuente: elaboración propia).*

Figura 4.1 Representación de datos curvilíneos *(Fuente: elaboración propia).*

de puntos, los cuales pueden representar secciones de tipo poligonal, o bien curvas. La condiciones de qué tipo de elemento representan estos puntos de control, vienen dados por un segundo vector denominado vector de puntos de interpretación.

4.3. *Clipping* sobre beziergons

La solución al problema de *clipping* sobre beziergons, en adelante *clipping*-b, que ha sido propuesta en este trabajo, toma ideas tanto del algoritmo de Sutherland-Hodgman como de Weiler-Atherton. Esto es, mediante la idea de Sutherland-Hodgman de extender infinitamente las aristas del agente, se logra reducir el problema de *clipping* entre dos geometrías generales (agente y sujeto), a encontrar *clipping* entre un semiplano (agente) y el sujeto, siendo el último de estos problemas resuelto mediante Weiler-Atherton. Este enfoque permite simpli-

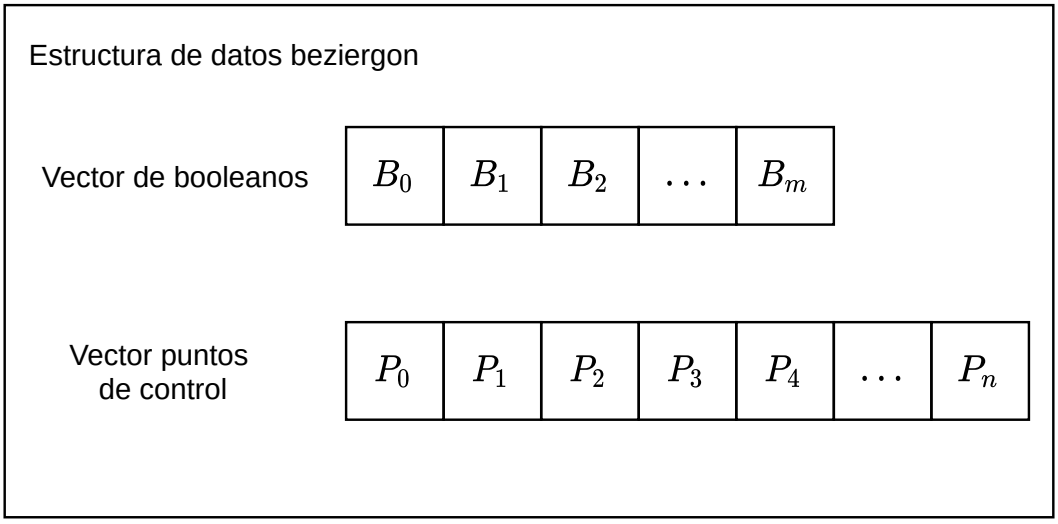


Figura 4.2 Representación estructura de datos propuesta *(Fuente: elaboración propia)*.

ficar la implementación del algoritmo, tal como lo haría Sutherland-Hodgman, pero sin perder las ventajas de Weiler-Atherton.

4.3.1. Preliminares

El algoritmo propuesto basado en beziergons, requiere definir las mismas reglas que la versión de polígonos, para lo cual es necesario etiquetar el tipo de intersección, esto es, entrante o saliente a un semiplano, pero ahora para intersecciones que involucren segmentos o curvas (en adelante *objeto*).

Para poder etiquetar las intersecciones como entrante o saliente el algoritmo *clipping*-b, requiere definir 2 elementos, el primero de ellos son los tipos de *interacciones* que existen entre segmento/recta infinita y curva/recta infinita. El segundo elemento corresponde la idea de *estado*, el cual toma valores **inside** o **outside**, dependiendo si el *objeto* actual está dentro o fuera del semiplano de interés. Notar que el *estado* se modifica dependiendo del tipo de *interacción* que se produce entre los elementos del sujeto y agente, como se mostrará más adelante.

Las interacciones entre segmento/recta infinita y curva/recta infinita son las dadas acontinuación. Notar que solo se resuelve el problema de intersección respecto recta infinita, debido a que el algoritmo solo requiere resolver esa condición.

a) Interacción segmento con recta infinita, figura 4.3:

1) Intersección.
2) Intersección con un punto extremo.
3) Colinealidad.
4) Sin intersección.

b) Interacción curva con recta infinita:

1) Caso 1: sin intersección, figura 4.4.
2) Caso 2: solo una intersección que no está en un punto de control extremo, figura 4.4, figura 4.4.
3) Caso 3: una intersección con un punto de control extremo, figura 4.5.
4) Caso 4: dos intersecciones, las cuales no están en los puntos de control externos, figura 4.6.
5) Caso 5: dos intersecciones, donde una ocurre en un punto de control externo, figura 4.7.
6) Caso 6: dos intersecciones, que ocurren exactamente en los puntos de control extremos, figura 4.8.
7) Caso 7: intersección tangencial, figura 4.8.

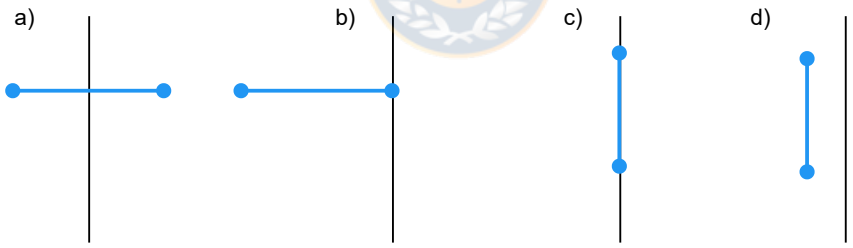


Figura 4.3 Interacciones entre un segmento y una recta infinita *(Fuente: elaboración propia)*.

Por otro lado, el *estado* cambia de la siguiente forma, considerando que el *estado* toma como valor inicial **outside**:

a) Cambios de *estado* cuando interactúa un segmento con una recta infinita, figura 4.3:

1) Intersección: **outside** $\longrightarrow$ **inside**.
2) Intersección con un punto extremo: al *estado* se le asigna un valor del estado del segment actual, es decir, se pregunta si el segmento está dentro o fuera del semiplano.

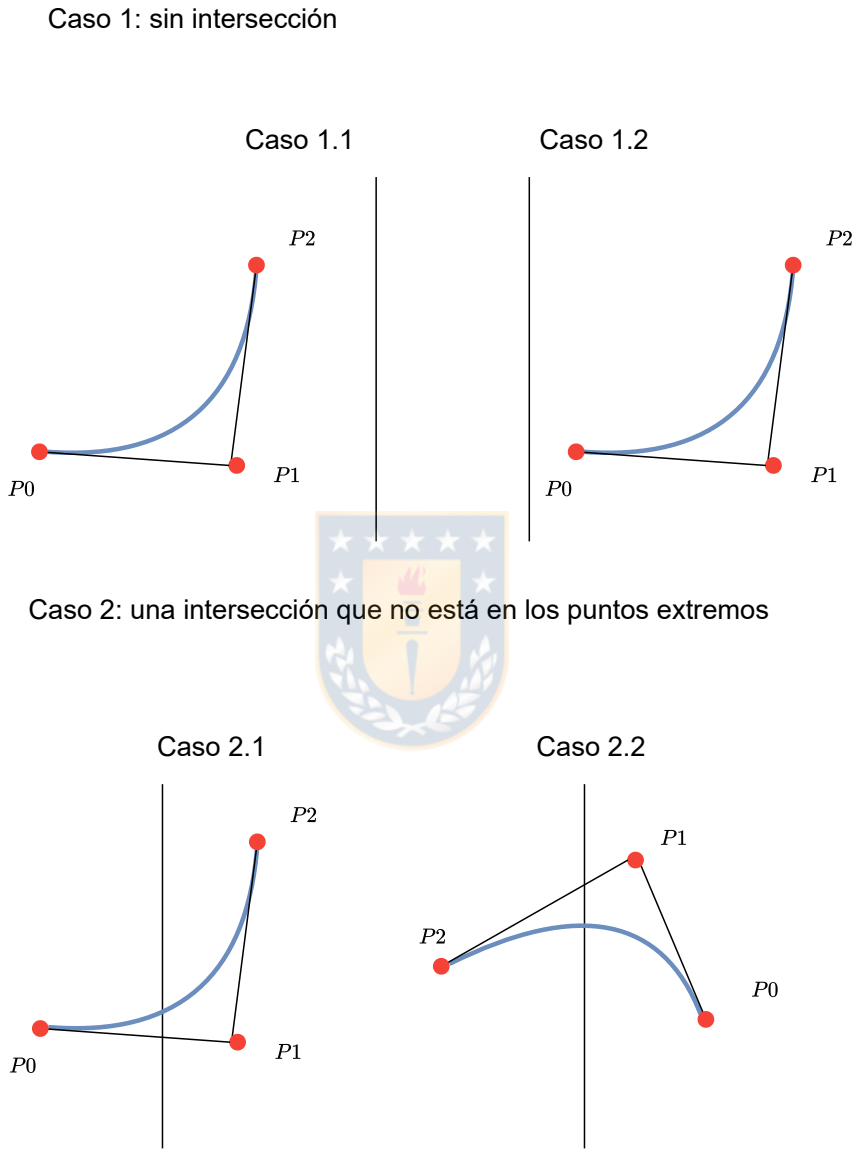


Figura 4.4 Caso 1: sin intersección, caso 2: intersección simple que no está en puntos extremos *(Fuente: elaboración propia)*.

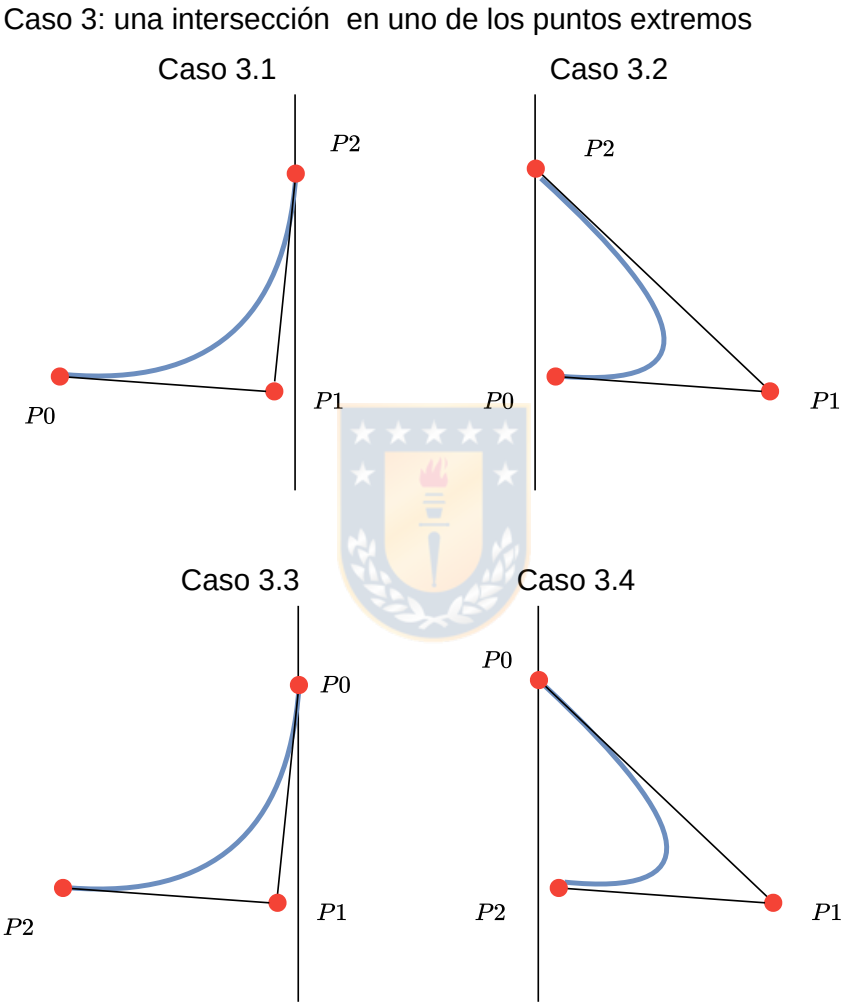


Figura 4.5 Caso 3: intersección simple y en un punto extremo *(Fuente: elaboración propia)*.

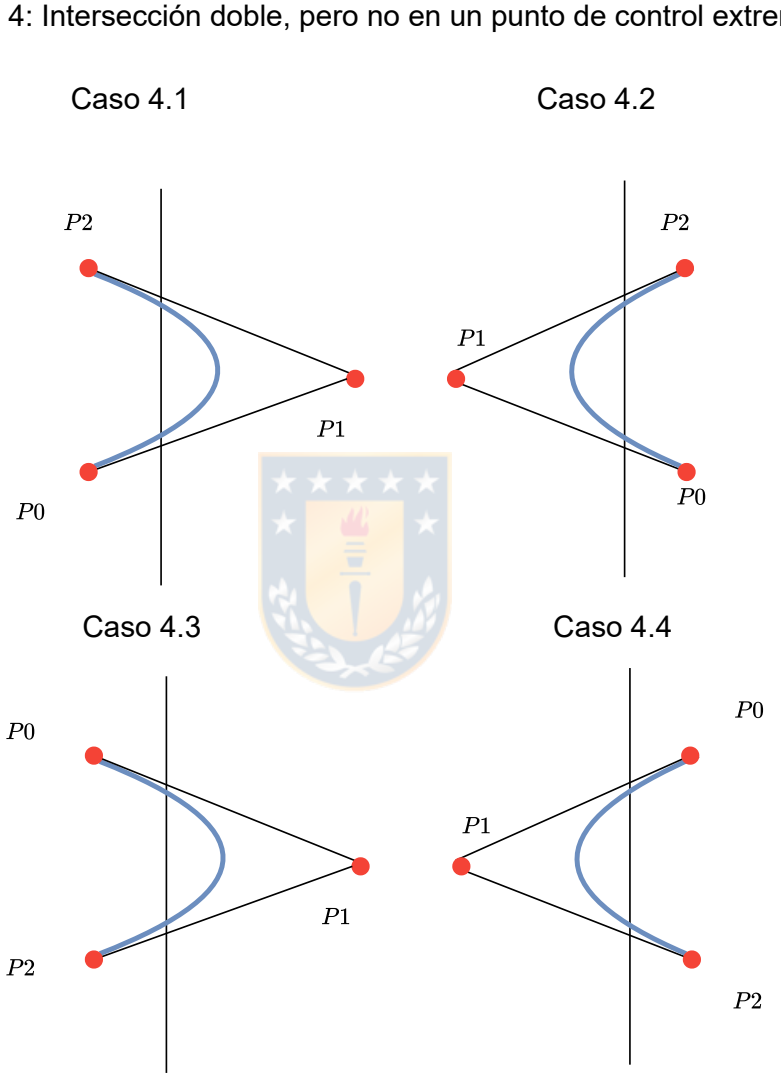


Figura 4.6 Caso 4: intersecciones dobles, pero no en puntos extremos *(Fuente: elaboración propia)*.

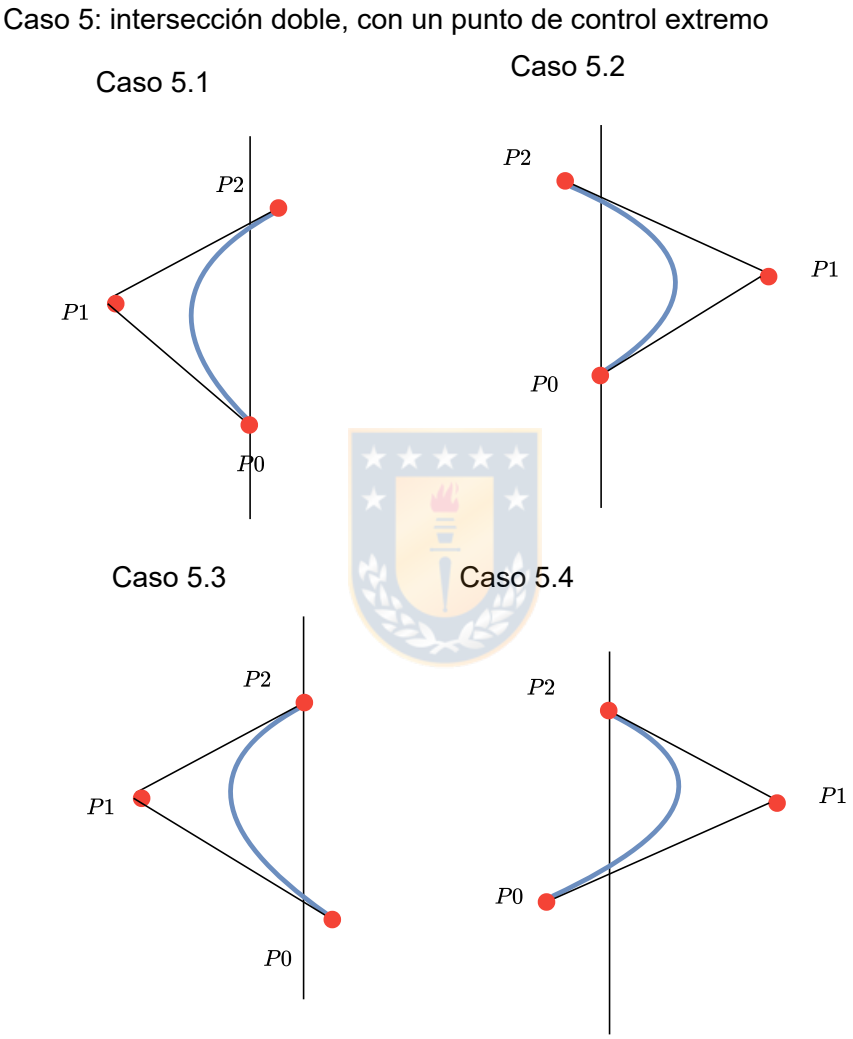


Figura 4.7 Caso 5: intersecciones dobles, con una intersección en un punto extremo *(Fuente: elaboración propia)*.

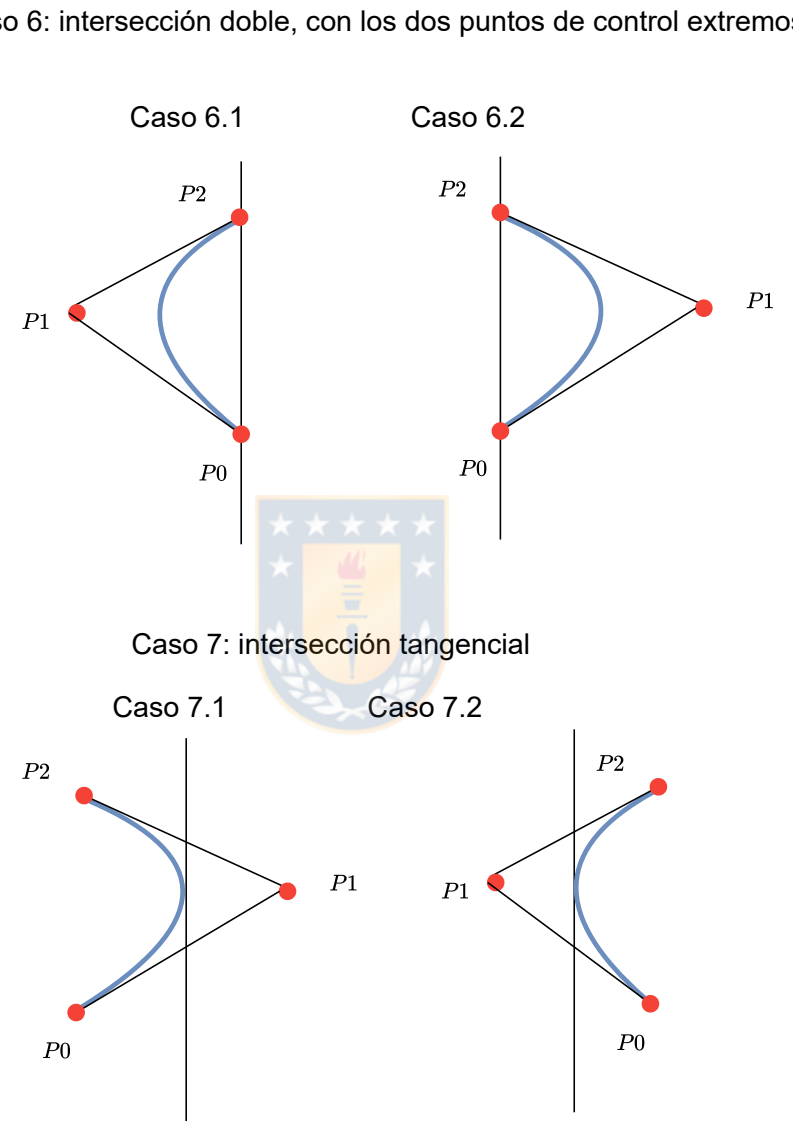


Figura 4.8 Caso 6: intersecciones dobles, en puntos extremos, caso 7: intersecciones tangenciales *(Fuente: elaboración propia)*.

3) Colinealidad: al *estado* se le asigna el valor del estado del segmento actual.
4) Sin intersección: no cambia.

b) Cambios de *estado* cuando interactúa una curva con recta infinita, se peden ver en las figuras 4.4 a 4.8:

1) Intersección simple: **outside** $\longrightarrow$ **inside**.
2) Intersección doble: al *estado* se le asigna valor del *estado* que tiene la subdivisión izquierda de la curva actual.
3) Intersección con un punto de control extremo: al *estado* se le asigna un valor del estado de la curva actual.
4) Intersección tangencial: no cambia.
5) Sin intersección: no cambia.

Descripciones más detalladas de todas los tipos de interesecciones relacionadas con las curvas se pueden encontrar en los algoritmo 5, algoritmo 6, algoritmo 7.

4.3.2. Descripción de *clipping*-beziergon

Habiendo definido una estrategia para etiquetar las intersecciones de forma consistente, es posible analizar el algoritmo propuesto. *Clipping*-b se compone de dos procedimientos:

a) Generar *geometrías parciales*, la cual efectúa 2 tareas:

1) Generar un conjunto de geometrías comprendidas entre un punto de intersección entrante y uno saliente, de forma de cumplir con la regla 1.
2) Generar un vector de intersecciones entre el agente (un semiplano) y el sujeto, de modo de satisfacer la regla 2.

b) Unión: consiste en unir las geometrías parciales del paso anterior, de modo de retornar geometrías cerradas.

Por lo tanto, el paso 1 de *clipping*-b: *geometrías parciales* realiza las tareas listadas más abajo, y se detalla en los algoritmos 3, 4 y 5.

a) Recibir como entrada una geometría almacenada en una estructura de datos beziergon, y una coordenada que representa una recta vertical.

b) Inicializar un iterador circular sobre el beziergon de entrada.

c) Busca la primera intersección de tipo entrante como entrante.

d) Itera cada uno de los elementos del beziergon, y aplica las *interacciones* descritas en la sección anterior de forma diferenciada para segmento y curvas, las cuales corresponden a las subrutinas 4 y 5, respectivamente. La idea clave para generar las geometrías parciales, es almacenar todo lo que se encuentre entre una intersección entrante hasta que se encuentre una intersección saliente, para luego comenzar a generar una nueva gometría parcial con el mismo procedimiento.

e) Retorna un vector de geometrías parciales, y un vector ordenado que almacena todas las intersecciones entre el beziergon de entrada y la recta infinita.

Por otro lado, *clipping*-b: *unión* se compone de los siguientes pasos:

a) Recibir un vector de geometrías parciales, y un vector que almacene intersecciones del procedimiento anterior.

b) Por cada geometría parcial agregar a su extremo final, un segmento que va desde el punto del extremo final de dicha geometría y el punto inmediatamente superior dentro del vector de intersecciones.

c) Finalmente unir todas las geometrías parciales cuyos puntos de inicio-fin, fin-inicio sean coincidentes.

Algoritmo 3: *Clipping*-b geometrías parciales

```
   Input: (beziergon, xVertical)
   Output: (bufferPartialBeziergons, pointIntersectionVector)
1  curveOrSegment = CircularIterator(Beziergon)
   /* Move to the first incomming intersection event                    */
2  moveToTheFirstIncomingIntersection(beziergon, curveOrSegment, xVertical)
3  bool status = false
4  Integer curveOrSegmentCurrent = 0
5  Beziergon partialBeziergon
   /* Iterate over the beziergon object                                 */
6  while curveOrSegmentCurrent ⩽ beziergon.size() do
       /* Segment case                                                  */
7      if curveOrSegment.isSegment() then
8          segmentProcessor(curveOrSegment, xVertical, status,
             bufferPartialBeziergons, bufferPartialBeziergons)
       /* Curve case                                                    */
9      else
10         curveProcessor(curveOrSegment, xVertical, status, bufferPartialBeziergons,
             bufferPartialBeziergons)
11     ++curveOrSegmentCurrent
12     ++curveOrSegment
13 return (bufferPartialBeziergons, PointIntersectionVector)
```

Algoritmo 4: *Clipping*-b geometrías parciales, procesador de segmentos

```
   Input: (curveOrSegment, xVertical, reference status, reference
           partialBeziergon,reference bufferPartialBeziergons)
1  segmentEvent = IntersectionRespectVertical(circularIterator,xVertical)
   /* Detect intersection but not with the external control points          */
2  if segmentEvent.pointIntersection then
       /* Switch the state, due there is a state change                     */
3      status = !status
       /* If there is an incoming intersection                              */
4      if status then
5          partialBeziergon.appendSegment(segmentEvent.intersectionPoint,curveOrSegment.head())
       /* If there is an outgoing intersection                              */
6      else
7          partialBeziergon.appendSegment(curveOrSegment.tail(),
             segmentEvent.intersectionPoint)
8          bufferPartialBeziergons.push_back(partialBeziergon)
9          partialBeziergon.clear()
10     pointIntersectionVector.push_back(segmentEvent.intersectionPoint)
   /* Detect intersection with the external control points                  */
11 else if segmentEvent.externalControlPointIntersection then
12     status = statusOfLeftHalfPlane(curveOrSegment, xVertical);
13     if status then
14         partialBeziergon.appendSegment(curveOrSegment))
           /* If there is an incoming intersection                          */
15         if segmentEvent.tailIntersection() then
16             pointIntersectionVector.push_back(segmentEvent.tailIntersection)
           /* If there is an outgoing intersection                          */
17         if segmentEvent.headIntersection() then
18             pointIntersectionVector.push_back(segmentEvent.headIntersection)
19             bufferPartialBeziergons.push_back(partialBeziergon)
20             partialBeziergon.clear()
   /* Without intersection                                                 */
21 else
22     if status then
23         partialBeziergon.appendSegment(curveOrSegment))
24 return
```

Algoritmo 5: *Clipping*-b geometrías parciales, procesador de curvas

```
   Input: (curve, xVertical, reference status, reference partialBeziergon,reference bufferPartialBeziergons)
1  curveEvent = IntersectionRespectVertical(circularIterator,xVertical)
   /* Curve intersection event, not tangential,not with the extreme control
      points                                                                  */
2  if curveEvent.intersection then
       /* One intersection case                                              */
3      if curveEvent.centerCurve == none then
4          simpleIntersectionCurveInfiniteLine()
       /* Two intersection case                                              */
5      else
6          doubleIntersectionCurveInfiniteLine()
   /* Extreme control point intersection                                      */
7  else if curveEvent.externalControlPointIntersection then
8      status = statusOfLeftHalfPlane(curve.leftCurve, xVertical)
9      if status then
10         partialBeziergon.appendCurve(curveEvent.curve)
11         pointIntersectionVector.push_back(curveEvent.curve.finalPoint)
12         pointIntersectionVector.push_back(curveEventcurve..finalPoint)
13         bufferPartialBeziergons.push_back(partialBeziergon)
14         partialBeziergon.clear()
   /* Tangential intersection case                                            */
15 else if curveEvent.tangentialIntersection and status then
16     if status then
17         partialBeziergon.appendCurve(curveEvent.leftCurve))
18         pointIntersectionVector.push_back(curveEvent.leftCurve.finalControlPoint)
19         bufferPartialBeziergons.push_back(partialBeziergon)
20         partialBeziergon.clear()
21         partialBeziergon.appendSegment(curveEvent.rightCurve))
   /* Without intersection                                                    */
22 else
23     if status then
24         partialBeziergon.appendCurve(curve))
25 return
```

Algoritmo 6: Intersección simple curve-recta infinita

```
   Input: (curve, xVertical, reference status, reference partialBeziergon,reference bufferPartialBeziergons)
   /* Intersecton with no extreme control points                                   */
1  if curveEvent.LeftPoint == none and curveEvent.RightPoint == none then
2      status = !status
3      if status then
4          partialBeziergon.appendCurve(curveEvent.rightCurve)
5          pointIntersectionVector.push_back(curveEvent.rightCurve.initialControlPoint)
6      else
7          partialBeziergon.appendCurve(curveEvent.leftCurve)
8          pointIntersectionVector.push_back(curveEvent.rightCurve.finalControlPoint)
             bufferPartialBeziergons.push_back(partialBeziergon)
9          partialBeziergon.clear()
   /* Intersecton with control points                                              */
10 else
11     if curveEvent.curve.rightPoint ≤ xVertical and curveEvent.curve.leftPoint ≤ xVertical then
12         status = true
           /* Intersecton with left control point                                  */
13         if curveEvent.leftPoint ! = none and curveEvent.rightPoint == none then
14             partialBeziergon.appendCurve(curveEvent.curve)
15             pointIntersectionVector.push_back(curveEvent.curve.leftPoint)
           /* Intersecton with right control point                                 */
16         else
17             status = false
18             partialBeziergon.appendCurve(curveEventC.urve)
19             pointIntersectionVector.push_back(curveEvent.curve.rightPoint)
20             bufferPartialBeziergons.push_back(partialBeziergon)
21             partialBeziergon.clear()
22 return
```

Algoritmo 7: Intersección doble curve-recta infinita

```
   Input: (curve, xVertical, reference status, reference partialBeziergon,reference bufferPartialBeziergons)
   /* Two intersection, but not with external control points                    */
1  if curveEvent.leftPoint == none and curveEvent.leftPoint == none then
2      status = statusOfLeftHalfPlane(curveEvent.leftCurve, xVertical)
       /* Two intersection without extreme, inside                             */
3      if status then
4          partialBeziergon.appendSegment(curveEvent.leftCurve))
5          pointIntersectionVector.push_back(curveEvent.leftCurve.finalControlPoint)
             bufferPartialBeziergons.push_back(partialBeziergon)
6          partialBeziergon.clear()
7          partialBeziergon.appendSegment(curveEvent.rightCurve))
8          pointIntersectionVector.push_back(curveEvent.rightCurve.initialControlPoint)
       /* Two intersection without extreme, outside                            */
9      else
10         partialBeziergon.appendSegment(curveEvent.centerCurve))
11         pointIntersectionVector.push_back(curveEvent.centerCurve.initialControlPoint)
             pointIntersectionVector.push_back(curveEvent.centerCurve.finalControlPoint)
             bufferPartialBeziergons.push_back(partialBeziergon)
12         partialBeziergon.clear()
   /* Two intersection, one of them is a external control point                */
13 else
14     if curveEvent.leftPoint ! = none and curveEvent.leftPoint == none then
15         status = statusOfLeftHalfPlane(curve.leftCurve, xVertical)
           /* Intersection with LeftPoint, centerCurve inside                  */
16         if status then
17             status = false
18             partialBeziergon.appendSegment(curveEvent.centerCurve))
19             pointIntersectionVector.push_back(curveEvent.centerCurve.initialControlPoint)
                 pointIntersectionVector.push_back(curveEvent.centerCurve.finalControlPoint)
                 bufferPartialBeziergons.push_back(partialBeziergon)
20             partialBeziergon.clear()
           /* Intersection with LeftPoint, centerCurve outside                 */
21         else
22             status = true
23             partialBeziergon.appendSegment(curveEvent.rightCurve))
24             pointIntersectionVector.push_back(curveEvent.rightCurve.initialControlPoint)
25     if curveEvent.leftPoint == none and curveEvent.leftPoint ! = none then
26         s
27     tatus = statusOfLeftHalfPlane(curve.leftCurve, xVertical)
       /* Intersection with RightPoint, centerCurve inside                     */
28     if status then
29         status = false
30         partialBeziergon.appendSegment(curveEvent.centerCurve))
31         pointIntersectionVector.push_back(curveEvent.centerCurve.initialControlPoint)
             pointIntersectionVector.push_back(curveEvent.centerCurve.finalControlPoint)
             bufferPartialBeziergons.push_back(partialBeziergon)
32         partialBeziergon.clear()
       /* Intersection with RightPoint, centerCurve inside                     */
33     else
34         status = false
35         partialBeziergon.appendSegment(curveEvent.leftCurve))
36         pointIntersectionVector.push_back(curveEvent.leftCurve.finalControlPoint)
37         ufferPartialBeziergons.push_back(partialBeziergon)
38         partialBeziergon.clear()
39 return
```

4.3.3. *Clipping* de ventana

Extender *clipping*-b entre una geometría cóncava o convexa como sujeto con una cóncava como agente, resulta sencillo, al rotar apropiadamente al sujeto. El caso particular de *clipping* de ventana, el sujeto se debe rotar 4 veces tal como lo indica el algoritmo 8.

Algoritmo 8: Window clipping

Input: (BeziergonBuffer, windowClipping)
Output: BeziergonBufferClipped

```
/* Applies left clipping                                          */
BeziergonBufferClipped = LeftClipping(BeziergonBuffer,
  windowClipping.rightCoordinate)
/* Rotate the geometeries by 90 degress                           */
rotateBy90(BeziergonBuffer)
/* Applies top clipping                                           */
BeziergonBufferClipped = LeftClipping(BeziergonBufferClipped,
  windowClipping.bottomCoordinate)
rotateBy90(BeziergonBuffer)
/* Applies right clipping                                         */
BeziergonBufferClipped = LeftClipping(BeziergonBufferClipped,
  windowClipping.leftCoordinate)
rotateBy90(BeziergonBuffer)
/* Applies bottom clipping                                        */
BeziergonBufferClipped = LeftClipping(BeziergonBufferClipped,
  windowClipping.topCoordinate)
/* Finally rtrive the original orientation                        */
rotateBy90(BeziergonBuffer)
return BeziergonBufferClipped
```

4.3.4. *Healing* sobre beziergons

Si bien, es posible considerar el algoritmo de *healing* basado en polígonos, como base para la construcción de uno basado en beziergons (en adelante *healing*-b), es necesario definir adaptar ciertos elementos para que funcione correctamente. Se requiere resolver los problemas de intersecciones de tipo segmento/segmento e intersección curva/curva, como se verá más adelante. Además, más dificultades se encuentran al tratar de definir elementos de *leading* y *trainling*, cuando de curvas se trata. Es por ello que por alcances de este trabajo se ha desarrollado sólo una implementación parcial del algoritmo de *healing*, que funciona para un conjunto particular de datos.

Resolver intersecciones

Adicionalmente al manejo de intersecciones segmento/segmento de *healing* de polígonos, *healing*-b debe resolver las intersecciones de tipo segmento/curva y curva/curva, las cuales pueden ser clasificadas según el número de objetos que retornan:

a) Intersección segmento/curva:
   1) 0 segmentos y 0 curvas de retorno, figura 4.9-a.
   2) 1 segmento y 1 curva de retorno, figura 4.9-b.
   3) 2 segmentos y 1 curva de retorno, figura 4.9-c.
   4) 3 segmentos y 1 curva de retorno, figura 4.9-d.
   5) 1 segmento y 2 curvas de retorno, figura 4.9-e.
   6) 2 segmentos y 2 curvas de retorno, figura 4.9-f.
   7) 2 segmentos y 3 curvas de retorno, figura 4.9-g.
   8) 3 segmentos y 3 curvas de retorno, figura 4.9-h.

b) Intersección curva/curva:
   1) 0 curvas de retorno, figura 4.9-a.
   2) 2 curvas de retorno, figura 4.9-b.
   3) 3 curvas de retorno, figura 4.9-c.
   4) 4 curvas de retorno, figura 4.9-d.
   5) 6 curvas de retorno, figura 4.9-e.

La solución propuesta para resolver este problema se basa en aplicar subdivisión, y aproximación de curvas mediante segmentos.

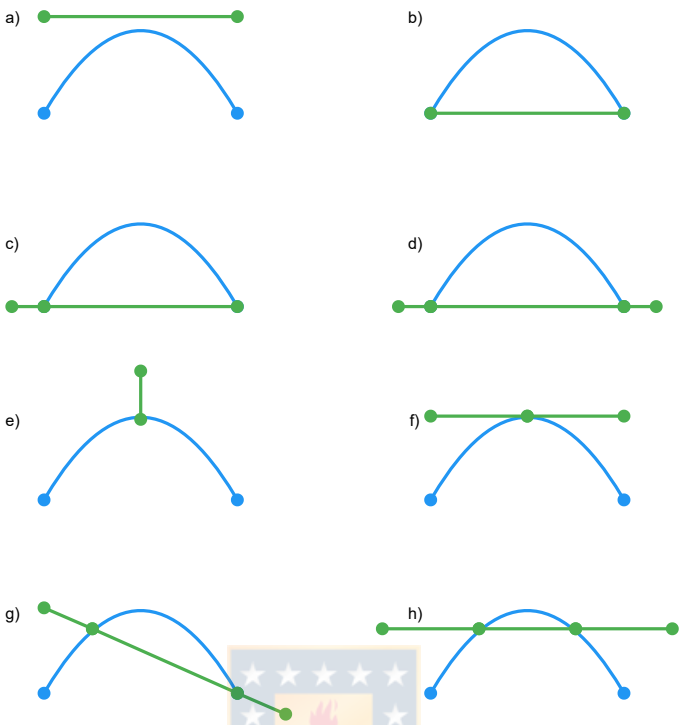


Figura 4.9 Eventos de intersección segmento/curva *(Fuente: elaboración propia)*.

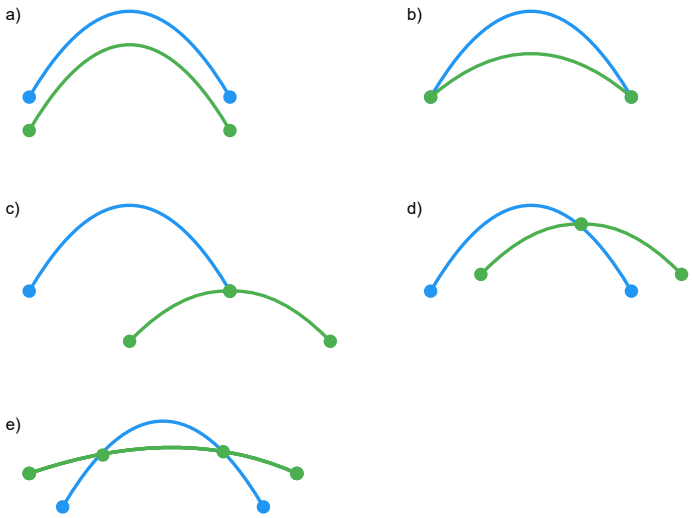


Figura 4.10 Eventos de intersección curva/curva *(Fuente: elaboración propia)*.

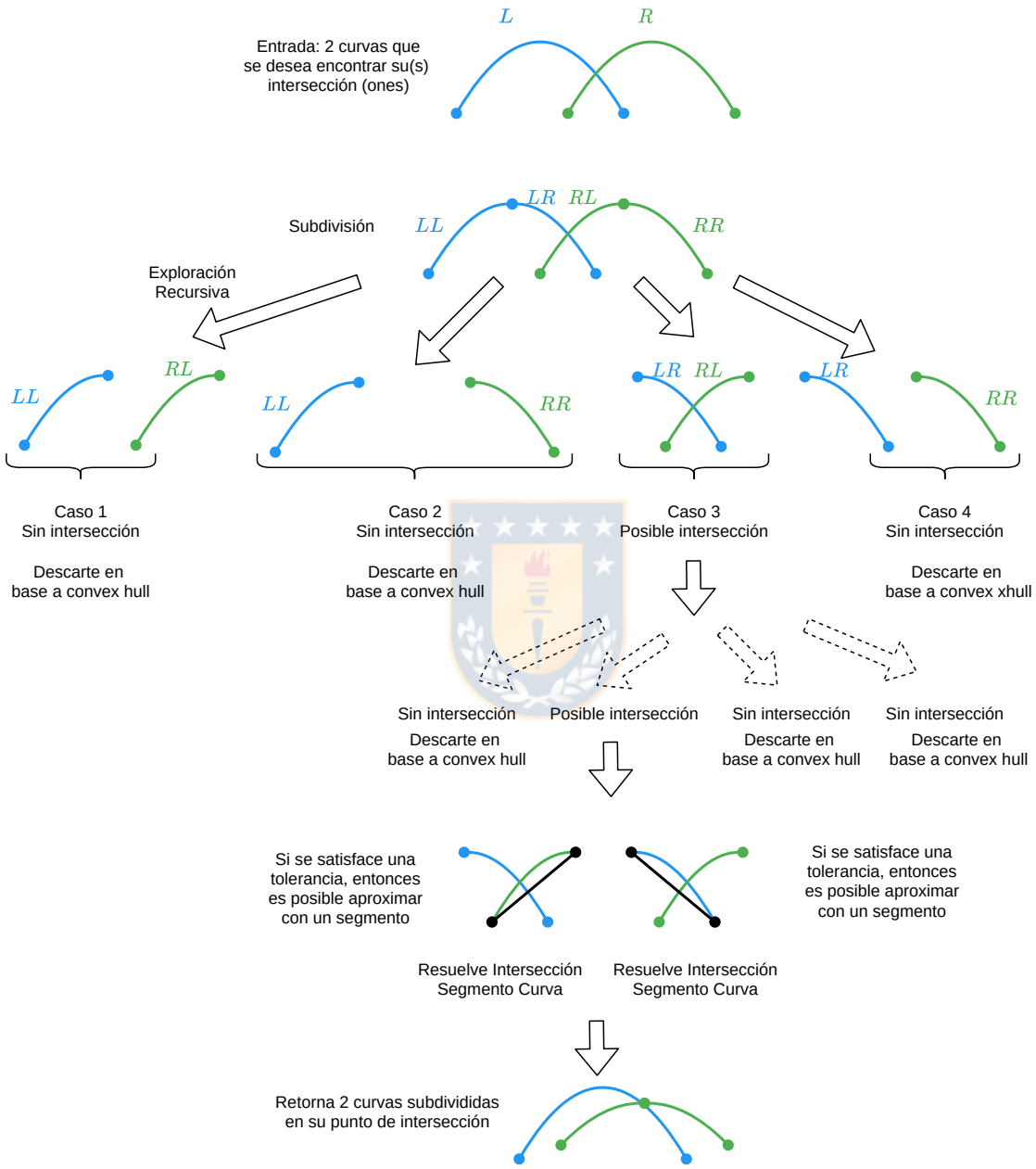


Figura 4.11 Estrategia de subdivisión y aproximación *(Fuente: elaboración propia)*.

Limitaciones

Si bien adaptar el algoritmo de *healing* del estado del arte, basado en una representación poligonal, a una representación basada en beziergon, resulta factible, hay que tratar adecuadamente ciertos conceptos, con tal de poder definir apropiadamente las invariantes que el algoritmo requiere.

Una de las limitaciones más relevantes, tiene que ver con poder decidir si una curva actúa como *leading* o *trailing*, como se aprecia en la figura 4.12, lamentablemente esta clasificación no es posible aplicarla directamente a curvas de Bézier. Esta limitación sobre la implementación del algoritmo de *healing* propuesta, se deja como trabajo futuro. Por otro lado, es posible comentar alguna de las posibles soluciones. Puede ser aplicada la subdivisión de las curvas de Bézier en todos los puntos donde tengan una tangente vertical (similar a lo que se hace con *clipping* para hacer que una curva sea función), de esta forma se garantiza que la horizontal que pase por ese punto genere un nuevo par de objetos de *leading* y *trailing*. Se deja como trabajo futuro la implementación y experimentación con dicha funcionalidad adicional.

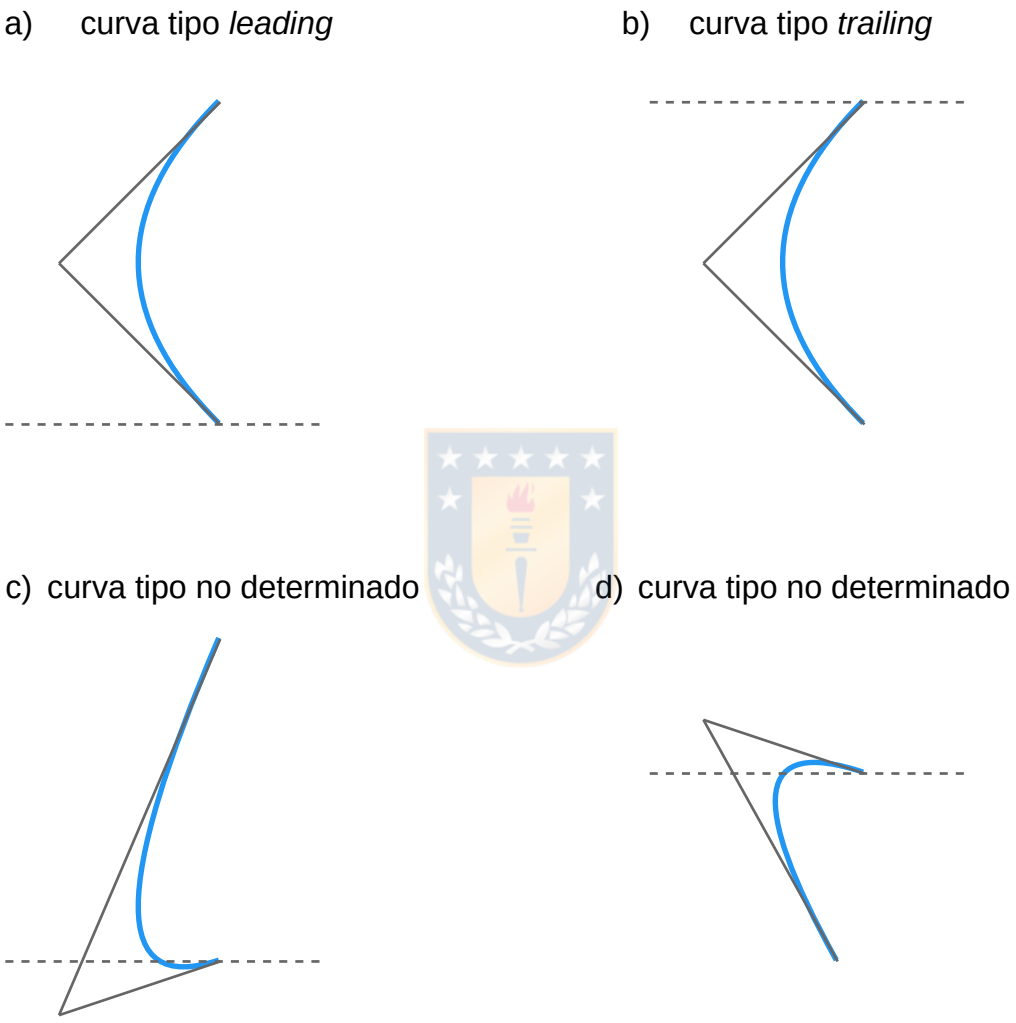


Figura 4.12 En las figuras a) y b) son survas de *leading* y *trailing*, respectivamente. Las figuras c) y d) no pueden clasificarse como *leading* o *trailing (Fuente: elaboración propia)*.

Capítulo 5

Experimentos y Resultados

5.1. Introducción

En los capítulos anteriores se ha explicado la representación propuesta en este trabajo, junto con los algoritmos y operaciones geométricas que trabajan con ella. En este capítulo se presentará el proceso experimental que estudia las implicancias en la introducción de curvas de Bézier, en en diseño en VLSI.

Se han elaborado un conjunto de experimentos, con el propósito de hacer un análisis en distintas métricas, tales como: el tiempo de ejecución, uso de memoria y calidad. Con propósitos comparativos, y para tener una referencia de base, es evidente que es necesario contar con al menos una representación adicional. En este caso resulta natural considerar la representación poligonal, actualmente utilizada por CATS. Con propósitos de hacer un análisis más profundo en, se ha decidido introducir artificialmente una tercera representación, que si bien no busca ser funcional, con ella se pretende obtener mayor cantidad de información para el análisis sobre el uso de curvas de Bézier respecto a una representación poligonal. Para ello se ha usado la característica de que la estructura de datos beziergon, permite el manejo de curvas de Bézier, como de segmentos. En dicho caso la tercera representación consiste en representar todos los datos exclusivamente mediante segmentos, evitando usar curvas, cuyo objetivo es hacer una comparación que extraiga las limitaciones intrínsecas a la implementación, haciendo que la comparación del uso de curvas versus segmentos se haga de manera más transparente.

El análisis de los resultados, conclusiones, limitaciones de la evaluación y trabajos futuros se abordan en el siguiente capítulo.

5.2. Experimentos

La arquitectura de cada configuración experimental, está constituida en términos generales de la siguiente forma:

a) Se reciben como datos de entrada las representaciones basadas en beziergons.

b) Los datos de entrada son escalados con el propósito de llevarlos a una resolución y tamaño usados en CATS.

c) Dependiendo del tipo de representación y del algoritmo que buscan evaluar, se deben transformar adecuadamente a beziergons o bien a polígonos.

d) Se aplica el algoritmo de interés y mide la métrica asociada.

e) Finalmente, es posible comparar resultados.

Así las representaciones con que se llevarán a cabo los experimentos son los siguientes:

a) Beziergons curvas: se basa en la estructura de datos beziergon y se caracteriza por usar curvas de bezier casi exclusivamente en la representación de los datos. De esta forma se pretende visualizar al máximo el comportamiento de los algoritmos usando dichas curvas.

b) Polígono: es la representación actual, usada en el proceso de MDP de Synopsys.

c) Beziergon segmentos: es una representación que está basada en beziergons, pero que se limita a almacenar los datos solo con segmentos, es decir, sin usar curvas de Bézier. La idea de esta representación es proveer al menos un criterio de comparación adicional, a la representación poligonal. Beziergon-segmentos no debe entenderse con un propósito funcional, sin más bien como herramienta de comparación adicional, a la representación poligonal, con la cual se busca extraer lo más posible las limitaciones de la implementación de la estructura de datos y/o algoritmos sobre beziergons, de esta forma los resultados de las ventajas en el uso de curvas de Bézier por sobre una representación poligonal pura, se hacen más evidentes.

Para los experimentos de *performance* y uso de memoria, se evalúan estas 3 representaciones, y se basan en la fuertemente en los pasos descritos anteriormente, y son los mostrados en las figuras 5.1 y 5.2. Sin embargo, notar que en el caso del experimento de uso de memoria figura 5.2, se ha separado el proceso de conversión de beziergons basados en curvas a polígonos, de tal forma que esta no influya en la lectura de memoria del experimento.

Por otro lado, la evaluación de calidad, figura 5.3, difiere un poco de los anteriores, en este caso solo se evalúan las representaciones de beziergon curvas y la representación poligonal, los cuales en este caso son contrastados con una representación poligonal, en alta resolución. Se debe entender como alta resolución a que el proceso de conversión de curvas de Bézier a polígonos se llevó a cabo con una precisión mayor.

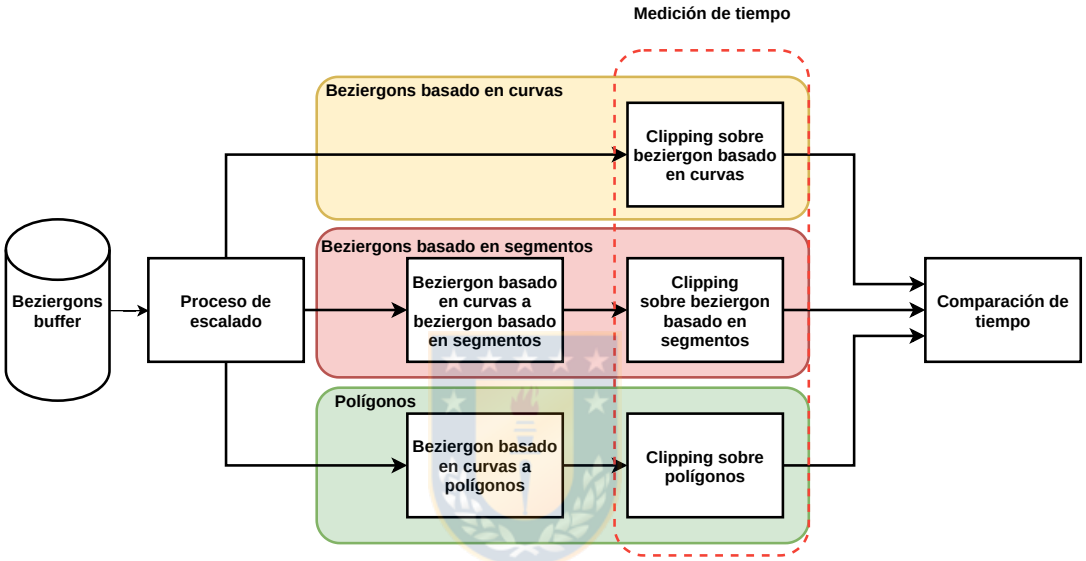


Figura 5.1 Configuración experimental *peformance (Fuente: elaboración propia)*.

5.3. Creación de datos artificiales

Por motivos de alcances de este trabajo, se ha elegido usar datos artificiales para efectuar las evaluaciones. La estrategia usada para elaborar dichos datos se basa en usar una plantilla, descrita en el apéndice B, en la cual se ubican ciertos puntos de control en forma pseudoaleatoria. Cabe destacar que algunos de los puntos de la plantilla se obtuvieron a partir de interpolación, con el propósito de que las curvas generadas sean suaves en sus uniones. Si bien, no es una restricción del dominio de los diseños en VLSI, en la práctica se aprecia esa tendencia.

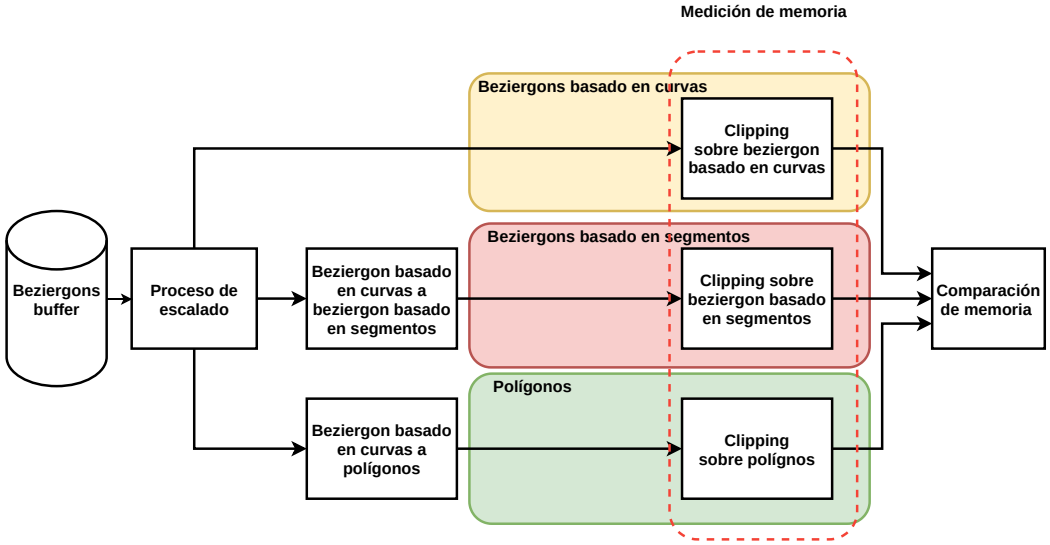


Figura 5.2 Configuración experimental uso de memoria *(Fuente: elaboración propia)*.

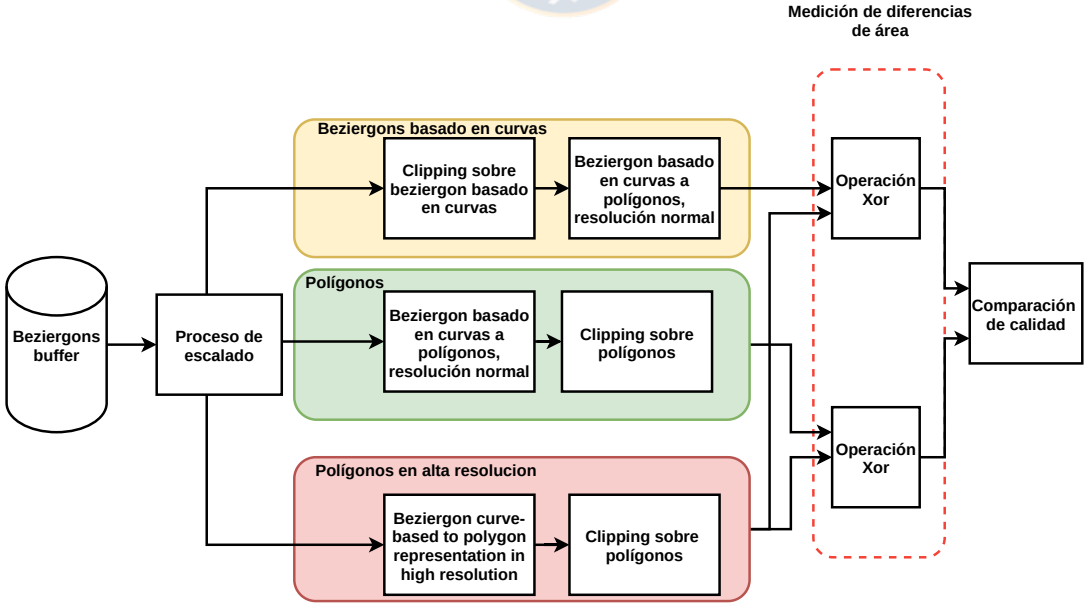


Figura 5.3 Configuración experimental calidad *(Fuente: elaboración propia)*.

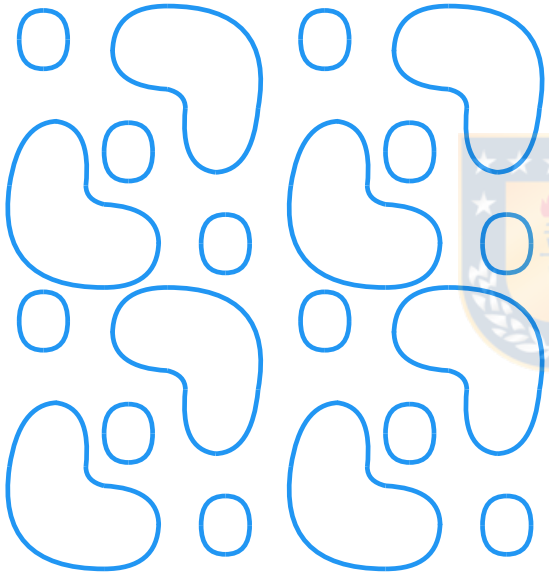
La forma dada a los patrones está basada fuertemente en la idea de que la representación resultante se ajuste lo más posible a datos reales. Las dos técnicas o tecnologías en las cuales se inspiró la forma de las plantillas son las siguientes:

a) Técnicas de fotolitografía inversa (ILT).

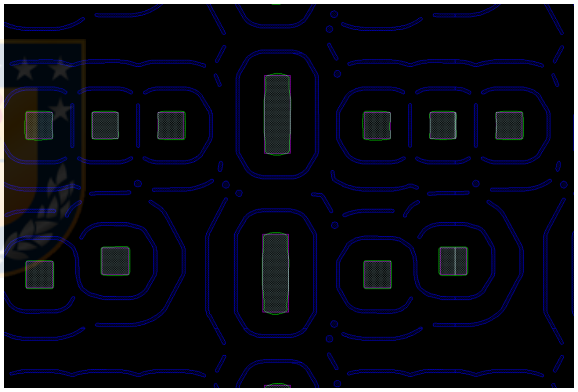
b) Tecnologías que estén basadas en luz, también denominadas ópticas en este trabajo.

Datos artificiales y reales correspondientes a la técnica ILT se muestra en las figuras 5.4-a y 5.4-b, lo mismo para las tecnologías ópticas se muestra en las figuras 5.5-a y 5.5-b.

Con ello es posible entregar los lineamientos para generar datos sintéticos con alguna semejanza a la de aplicaciones reales.

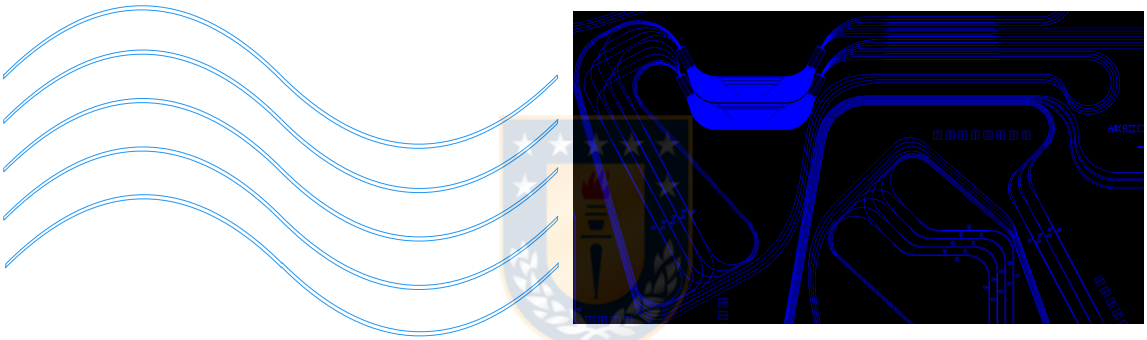


(a) Patrón ILT artificial *(Fuente: elaboración propia)*.



(b) Extracto real de un layout de tipo ILT *(Fuente: contribución de Synopsys)*.

Figura 5.4 Patrón ILT



(a) Patrón óptico artificial *(Fuente: elaboración propia).*

(b) Extracto real de un layout de tipo óptico *(Fuente: contribución de Synopsys).*

Figura 5.5 Patrón óptico

Geometría	Cantidad de Curvas	Cantidad Segmentos	Número de puntos layout
Beziergon curvas patrón ILT	960000	0	1920000
Beziergon segmentos patrón ILT	0	31289125	31289125
Polígono patrón ILT	0	31289125	31289125

Tabla 5.1 Descripción de datos artificiales ILT

Geometría	Cantidad de Curvas	Cantidad Segmentos	Número de puntos layout
Beziergon curvas patrón óptico	64000	32000	160000
Beziergon segmentos patrón óptico	0	81000000	81000000
Polígono patrón óptico	0	81000000	81000000

Tabla 5.2 Descripción de datos artificiales ópticos

5.4. Descripción de los datos artificiales

Una vez establecida la geometría de los patrones base de los artificiales, cabe señalar que estos se utilizaron para generar *layouts* más extensos, esto mediante la traslación de dicho patrón base. Dado que los patrones base tiene características pseudoaleatorias, es posible evitar que los algoritmos de CATS, exploten el uso de memoria caché. Esta es una carácteristicas deseable debido a que las implementaciones realizadas en este trabajo no tienen ese tipo de optimizaciones.

En las tablas 5.1 y 5.2 , se encuentra la información general de los layout utilizados. En ellas se indica el número curvas, segmentos y puntos de cada layout. De esta información la parte más relevante a considerar es número de puntos por *layout*, figura 5.6, dado que con ello es posible explicar el comportamiento de la mayoría de los experimentos abordados en las secciones siguientes.

5.4.1. Discretización curvas de Bézier

En este capítulo se hace bastante referencia al proceso de conversión de curvas de Bézier, a una representación a discretización. Este proceso es bastante simple, se trata de subdividir las curvas en forma recursiva, hasta satisfacer que la distancia entre la recta que une los puntos de control extremos de una curva de Bézier, y el punto de control intermedio, sea menor que cierta tolerancia, ver figura 5.7. Una vez conseguida la subdivisión de las curvas a la tolerancia deseada, basta aproximarlas por segmentos que unan los puntos de control extremos. Con este método es posible convertir datos desde la representación basada en beziergon, a una poligonal.

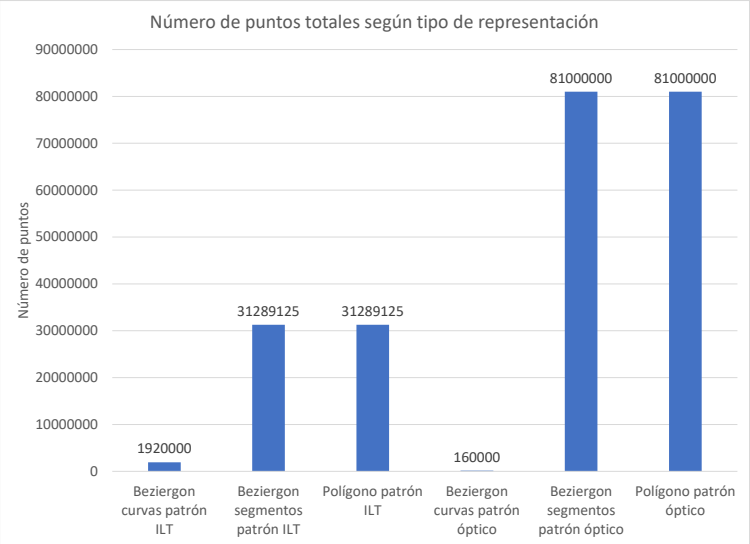


Figura 5.6 Número de puntos de cada representación y patrón usados en el experimento *(Fuente: elaboración propia)*.

Existen otras formas de discretizar las curvas de Bézier, pero este da garantías de que al aproximar las curvas resultantes por segmentos, dichas curvas estén confinadas dentro de la tolerancias dada.

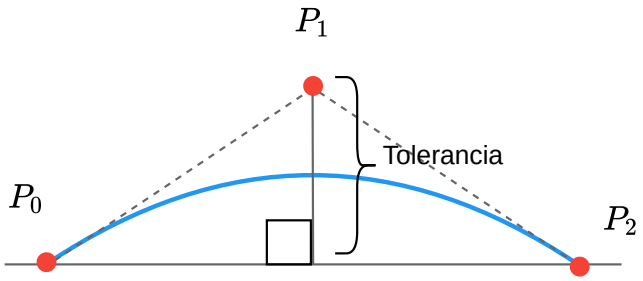


Figura 5.7 Concepto de tolerancia, con el cual puede discretizarse una curva. *(Fuente: elaboración propia)*.

5.5. Transformaciones Lineales

5.5.1. Performance

Los experimentos de *performance* en transformaciones lineales consisten aplicar la operación lineal en cuestión, habiendo aplicado un factor de traslación, rotación o escalado. El factor aplicado a cada transformación lineal se trata de un valor con distribución uniforme comprendido entre 1 y 100 que puede ser interpretado como una traslación en los ejes x e y, un factor de escalado o un ángulo de rotación, dependiendo de la transformación lineal. Así, se repitió 1000 veces la variación del parámetro, y se aplicó por cada tipo de representación y patrón. Los resultados *performance* de las seis representaciones están mostrados en la figura 5.8. Evidentemente que los resultados muestran que existe clara relación entre los tiempos requeridos al aplicar las operaciones lineales, con el número de puntos que contiene cada *layout*.

Los resultados de performance para las transformaciones las transformaciones lineales dan claros beneficios, en este caso para las tres transformaciones lineales en análisis, se obtiene que operar mediante representación propuesta resulta 10 veces más rápida que su símil poligonal, en el caso ILT, y de 361 veces más rápida, en el caso del patrón óptico.

5.5.2. Memoria

Los requerimientos de memoria RAM necesarios para aplicar las transformaciones lineales, se muestra en las figuras 5.9-a, 5.9-b y 5.9-c, para las operaciones de traslación, rotación y escalado, respectivamente. Notar que estas transformaciones no se han aplicado en forma *inplace*, razón por la cual es posible identificar en cada columna de los gráficos, una cantidad base necesaria para mantener los datos de entrada en memoria, y adicionalmente una porción que indica la memoria usada al aplicar la transformación lineal correspondiente. Notar además que existe distinta proporción entre la memoria requerida para mantener los datos de entrada y la memoria usada una vez aplicada la transformación lineal, las distintas representaciones ante la aplicación de la transformación lineal.

De forma similar a la *performance*, el uso de memoria guarda directa relación con la cantidad de puntos que contiene cada *layout* que se está operando.

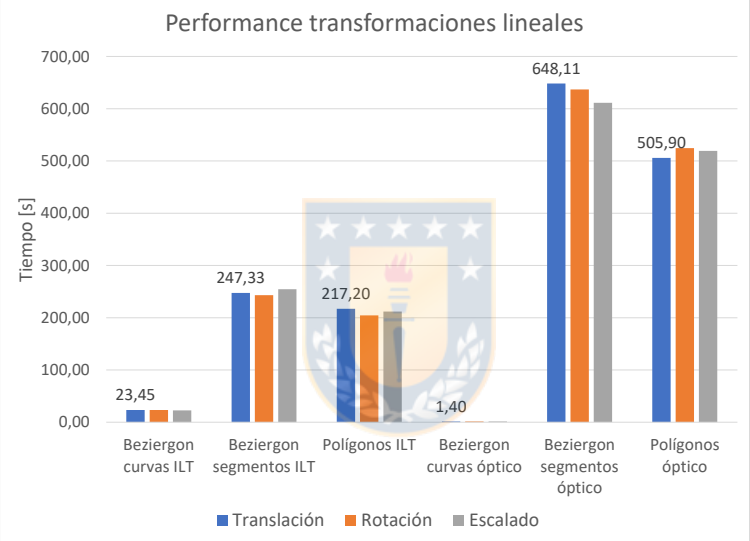
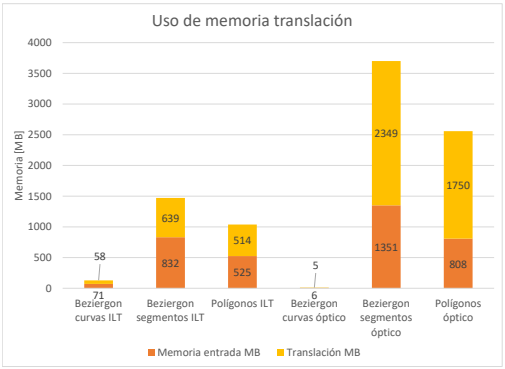
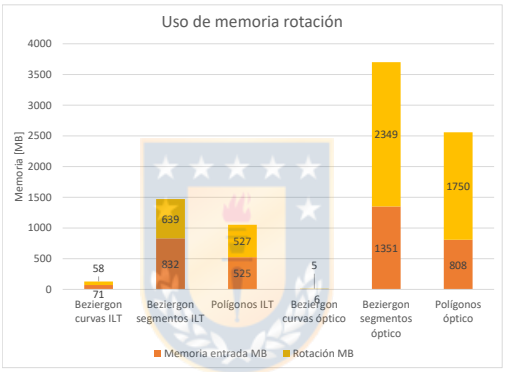


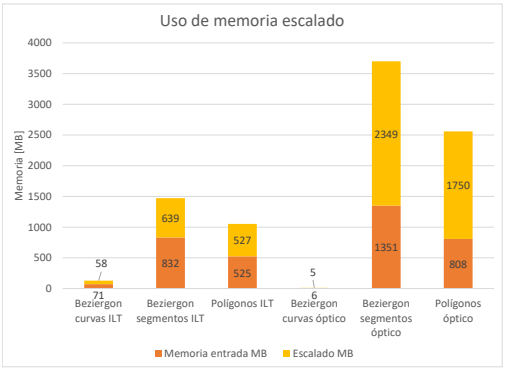
Figura 5.8 Resultado *performance* de las transformaciones lineales se translación, rotación y escalado, efectuada por cada representación de interés *(Fuente: elaboración propia)*.



(a) Transformación de traslación *(Fuente: elaboración propia)*.



(b) Transformación de rotación *(Fuente: elaboración propia)*.



(c) Transformación de escalado *(Fuente: elaboración propia)*.

Figura 5.9 Memoria usada por cada transformación lineal, y memoria requerida para almacenar los datos de entrada

5.6. Clipping

Los experimentos llevados a cabo para el algoritmo de *clipping* realizados en este trabajo, consistieron en posicionar una ventana de *clipping* cuadrada dentro del *layout* en forma pseudoaleatoria 1000 veces, y se fue tomando como entrada para el algoritmo de *clipping*. Este proceso fue repetido 10 veces, donde se fué incrementando el tamaño de la ventana de *clipping*.

5.6.1. Performance

En la tabla 5.3, se entregan los resultados de aplicar 1000 veces *clipping*, por cada tamaño de ventana de *clipping*, notar que por cada vez que se incremente el tamaño de la ventana de *clipping*, el algoritmo incrementa una cierta cantidad de tiempo, esto debido a que inicialmente el algoritmo considera un descarte por *bounding box*, por ende en la medida que la ventana de *clipping* incrementa su tamaño, el descarte por *bounding box* quita menos elementos de la geometría.

	Patrón ILT			Patrón óptico		
Lado ventana clipping cuadrada	**Beziergons curvas**	**Beziergon segmentos**	**Polígonos**	**Beziergons curvas**	**Beziergon segmentos**	**Polígonos**
2,50E+10	14,28	100,66	3,92	0,40	135,67	1,09
5,00E+10	14,28	101,21	4,01	0,41	136,84	0,97
7,50E+10	14,41	101,09	4,22	0,43	136,94	1,01
1,00E+11	14,41	101,58	4,41	0,43	135,58	1,07
1,25E+11	14,61	101,53	4,56	0,44	137,43	1,13
1,50E+11	14,69	102,18	4,73	0,44	137,76	1,17
1,75E+11	14,77	102,72	5,22	0,47	136,91	1,29
2,00E+11	14,83	102,40	5,50	0,46	136,97	1,29
2,25E+11	14,91	103,21	5,89	0,48	137,78	1,33
Promedio	**14,58**	**101,84**	**4,72**	**0,44**	**136,88**	**1,15**
Desviación estándar	**0,226**	**0,789**	**0,645**	**0,025**	**0,749**	**0,122**

Tabla 5.3 *Performance* algoritmo de *clipping*, 10 experimentos que contemplan 1000 ejecuciones de *clipping*.

Por otro, lado en la figura , se grafica el comportamiento promedio de cada tipo de representación, en las 10 veces que ser repitió el experimento, en cada uno de ellas se fue incrementando el tamaño de la ventana de *clipping*. Es posible apreciar que existe un ligero incremento en el tiempo del procesamiento del algoritmo, esto resulta concordante con el hecho de que el algoritmo tiene más datos que procesar con el incremento de la ventana de clipping, y puede descartar nemos elementos mediante *bounding box*.

Resulta relevante analizar qué partes de los algoritmos son más costosas en tiempo, en las figuras 5.11a, 5.11b y 5.11c, se indican las funciones más costosas para cada una de las representaciones estudiadas en para el conjunto de datos ILT, lo mismo aparece para el patrón óptico en las figuras 5.12a, 5.12b y 5.12c. En estos casos tanto para la representación prouesta y poligonal, las funciones con mayor costo computacional, tiene que ver con las relaciones de descarte de elementos mediante *bounding box*, siendo mayor al 93 % en el caso de la representación de beziergon curvas en el caso ILT y de 80 % en el caso del patrón óptico.

Otro antecedente a destacar, es que no parece ser excesivamente costoso los cálculos de intersección entre curvas y rectas. En un principio se pensó que dichas funciones podrían incurrir excesivo costo computacional.

5.6.2. Uso de memoria

Para el análisis del uso de memoria del algoritmo de *clipping*, se muestran los requerimientos de los valores máximo de memoria ram en la figura 5.13, este gráfico resulta prácticamente idéntico a los valores de uso de memoria de los datos de entrada, vistos en la sección de suso de memoria de las transformaciones lineales. Para entender los resultados de uso de memoria de algoritmos de clipping, cabe recordar que tanto la versión del algoritmos basado en biergon y la versión poligonal, realizan en una primera instancia descarte por *bounding box* de los elementos a operar. Siendo de esta forma solo una pequeña porción de los datos sometidos a operaciones de *clipping*.

5.6.3. Calidad

Los experimentos de calidad, consideran las representaciones de beziergons y polígonos para ambos patrones estudiados, los cuales se contrastan contra una

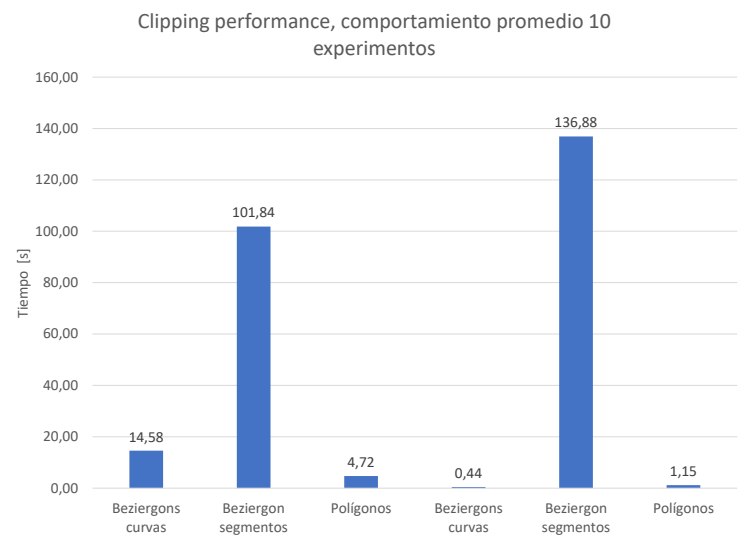


Figura 5.10 Comportamiento promedio de los experimentos de *clipping*, por cada tipo de representación estudiada *(fuente: elaboración propia)*.

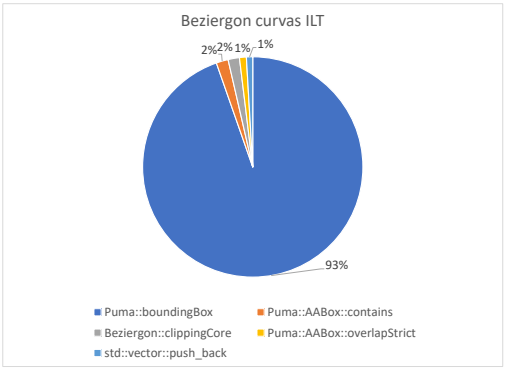
Representación	**Tolerancia normal**	**Tolerancia alta**
Beziergon patrón ILT	25000	250
Polígonos patrón ILT	25000	250
Beziergon patrón óptico	25000	2500
Polígonos patrón óptico	25000	2500

Tabla 5.4 Valores de tolerancia para convertir de curvas de Bézier a segmentos de recta.

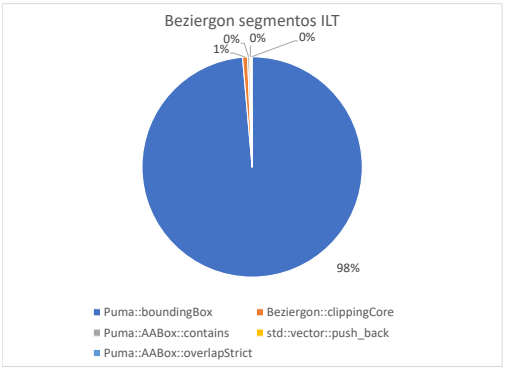
representación poligonal de alta calidad. Luego para determinar las diferencias entre el layout esperado y el estudiado, es necesario aplicar la operación xor entre estas dos representaciones. A las geometrías resultantes entonces es posible aplicarles alguna métrica que permita conocer su magnitud, en este trabajo se consideró la medida del área.

Debido a que en este proceso de medición de calidad, influyen fuertemente el proceso de conversión de las curvas de bezier, a una representación poligonal, es necesario considerar valores de tolerancias. Los valores de tolerancias usadas para las representaciones de polígonos y la de alta resolución varían en un factor de entre 100 y 1000. Así en la tabla 5.4 se muestran dichas cantidades.

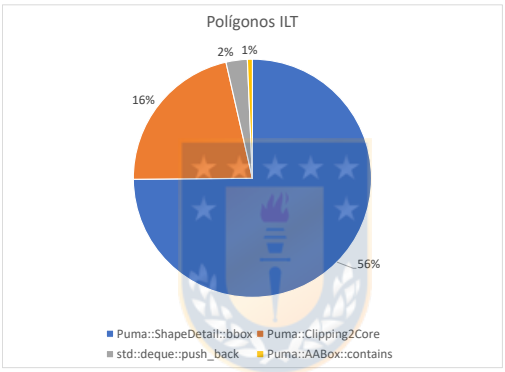
Una vez aplicada la operación xor entre las representaciones, es posible extraer bastante información, tal como se indica en los gráficos de las figuras 5.14-a



(a) Beziergons basados en curvas patrón ILT *(Fuente: elaboración propia).*



(b) Beziergons basados en segmentos patrón ILT *(Fuente: elaboración propia).*



(c) Polígonos patrón ILT *(Fuente: elaboración propia).*

Figura 5.11 Funciones más costosas de los algoritmos de *clipping* sobre el patrón ILT.

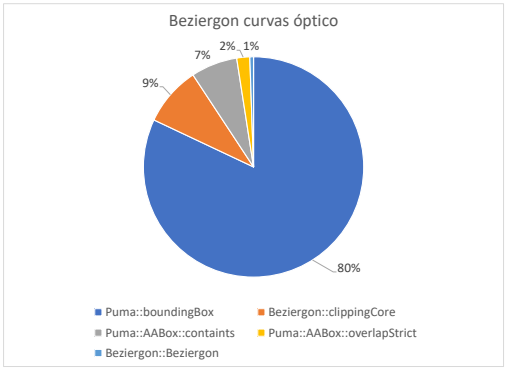
,5.15-a ,5.16-a y 5.20-a. En ellas es posible distinguir las diferencias de área para las representaciones siguientes:

a) Beziergons patrón ILT versus patrón esperado basado en polígonos de alta resolución.

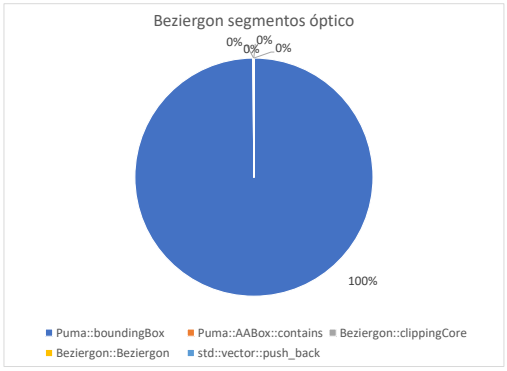
b) Polígonos patrón ILT versus patrón esperado basado en polígonos de alta resolución.

c) Beziergons patrón óptico versus patrón esperado basado en polígonos de alta resolución.

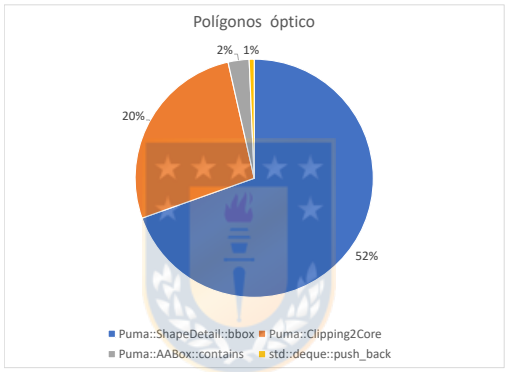
d) Polígonos patrón óptico versus patrón esperado basado en polígonos de alta resolución.



(a) Beziergons basados en curvas patrón óptico *(Fuente: elaboración propia).*



(b) Beziergons basados en segmentos patrón óptico *(Fuente: elaboración propia).*



(c) Polígonos patrón ILT *(Fuente: elaboración propia).*

Figura 5.12 Funciones más costosas de los algoritmos de *clipping* sobre el patrón óptico.

Además las diferencias se producen en los contornos de las figuras, lo cual es esperable debido a la aproximación de las representaciones.

Por otro lado es posible encontrar en las figuras 5.14-b, 5.15-b, 5.16-b y 5.20-b, una zona ampliada de estas diferencias de área.

Finalmente en las figuras figuras 5.14-c, 5.15-c, 5.16-c y 5.20-c, se muestran los histogramas que indican la cantidad de figuras en las que difieren las representaciones correspondientes, agrupadas según la magnituds de su área.

Ahora bien ,si se desea extraer la proporción entre el área total de diferencia entre las representaciones, y el área de las figuras que representan, entonces estas cantidades están indicadas en la tabla 5.5.

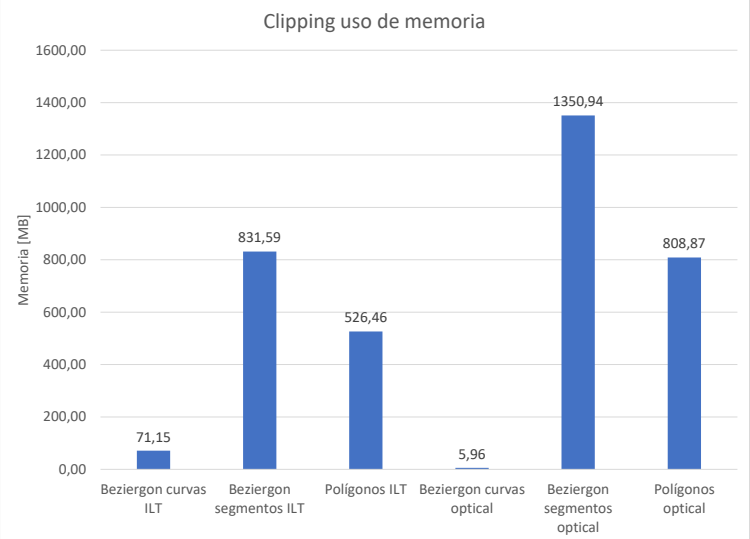


Figura 5.13 Uso de memoria de los experimentos de *clipping (Fuente: elaboración propia)*.

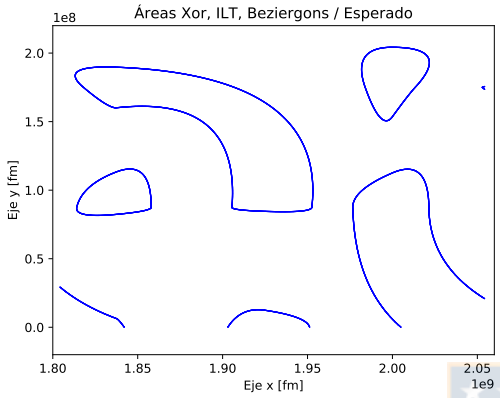
Representación	**Proporción de área.**
Beziergon curvas ILT	$\frac{4{,}98x10^{12}[fm^2]}{1{,}50x10^{15}[fm^2]} = 0{,}030\,\%$
Polígonos ILT	$\frac{4{,}72x10^{12}[fm^2]}{1{,}50x10^{15}[fm^2]} = 0{,}029\,\%$
Beziergon curvas óptico	$\frac{5{,}13x10^{12}[fm^2]}{1{,}72x10^{13}[fm^2]} = 29\,\%$
Polígonos óptico	$\frac{4{,}3610^{12}[fm^2]}{1{,}72x10^{13}[fm^2]} = 25\,\%$

Tabla 5.5 Proporciones entre las diferencias de área y el área de las geometerías que representan.

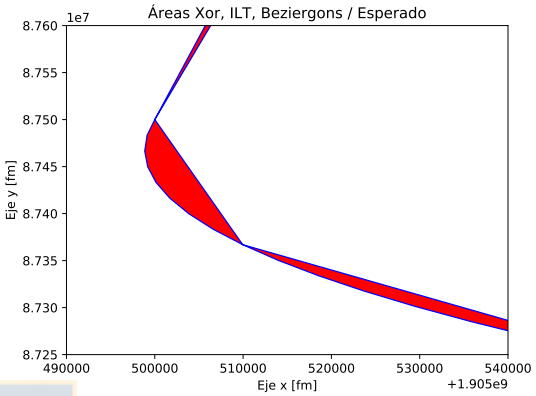
Notar que a menor cantidad de esta proporción indica mayor calidad, de esta lista son comparables las cantidades 1 y 2, y 3 con 4, puesto que de esta forma se están contrastando las representaciones propuesta en base a beziergons, y la actual en base a polígonos. Si bien en los casos 3 y 4 estas cantidades son de un 29 % y 25 %, esto se debe solamente a que en el caso del patrón óptico, el área que está delimitado por las figuras de este layout es pequeña, haciendo parece que el error es muy grande en estos casos.

Los gráficos mencionados anteriormente y de las proporciones calculadas previamente, parece ser que en cuanto a calidad las curvas de bézier, no tuvieron ventajas, sin embargo, hay que mencionar que para el proceso de conversión

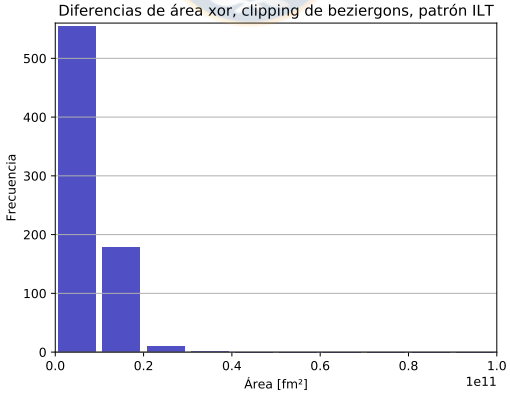
de las curvas de Bézier a la representación poligonal, se utilizó solamente una tolerancia igual a la usada para el caso de polígonos normal, por lo que todavía de ser necesario es posible aplicar una tolerancia más ajustada, con tal que las curvas de Bézier pueden ajustarse de mejor manera. En este último caso, se obtendrían ventajas significativas del uso de las curvas de Bézier por sobre la representación poligonal.



(a) Diferencias de área del patrón de beziergons ILT, con el patrón esperado *(Fuente: elaboración propia)*.

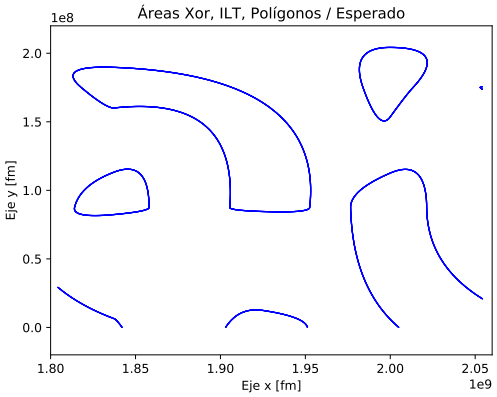


(b) Ampliación de una región del diseño, donde es posible apreciar las diferencias de área entre el patrón de beziergons ILT y el patrón esperado *(Fuente: elaboración propia)*.

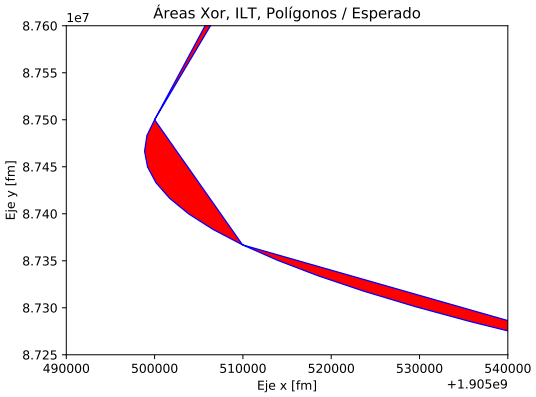


(c) Histograma de las diferencias de área entre el patrón Beziergon ILT y el patrón esperado *(Fuente: elaboración propia)*.

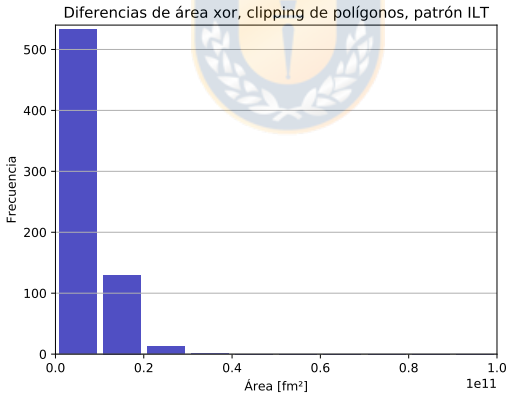
Figura 5.14 Gráficos de diferencias de área de la operación xor entre el patrón de beziergons ILT, y el patrón poligonal de alta resolución.



(a) Diferencias de área entre el patrón poligonal y el esperado *(Fuente: elaboración propia)*.

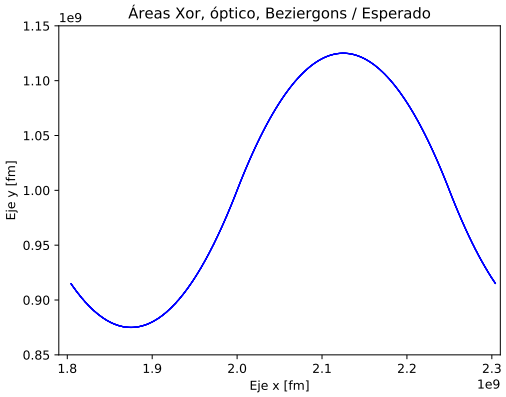


(b) Ampliación diferencias de área entre el patrón poligonal y el esperado *(Fuente: elaboración propia)*.

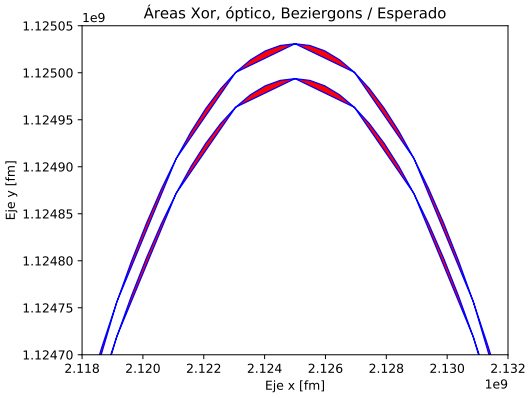


(c) Histograma de las diferencias de área entre el patrón poligonal ILT y el patrón esperado. Fuente elaboración propia.

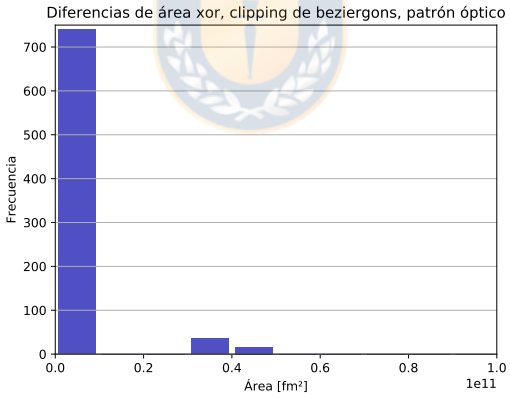
Figura 5.15 Gráficos de diferencias de área de la operación xor entre el patrón poligonal ILT, y el patrón poligonal de alta resolución.



(a) Diferencias de área beziergons y patrón esperado patrón óptico *(Fuente: elaboración propia).*

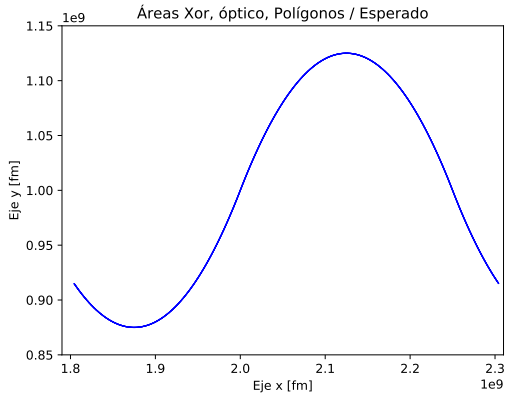


(b) Ampliación diferencias de área beziergons y patrón esperado patrón óptico *(Fuente: elaboración propia).*

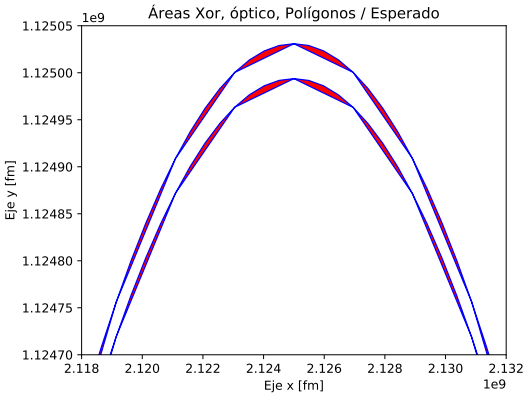


(c) Histograma de las diferencias de área entre el patrón beziergon óptico y el patrón esperado *(Fuente: elaboración propia).*

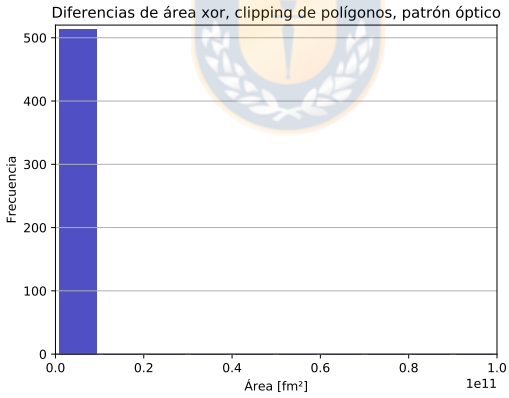
Figura 5.16 Gráficos de diferencias de área de la operación xor entre el patrón beziergons óptico, y el patrón poligonal de alta resolución.



(a) Diferencias de área polígonos y patrón esperado patrón óptico *(Fuente: elaboración propia)*.



(b) Ampliación diferencias de área polígonos y patrón esperado patrón óptico *(Fuente: elaboración propia)*.



(c) Histograma de las diferencias de área entre el patrón poligonal óptico y el patrón esperado *(Fuente: elaboración propia)*.

Figura 5.17 Gráficos de diferencias de área de la operación xor entre el patrón poligonal óptico, y el patrón poligonal de alta resolución.

Representación	Número de curvas	Número de segmentos	Total de puntos
Beziergon curvas	19208	0	38416
Beziergon segmentos	0	153664	307328
Polígonos	0	153664	307328

Tabla 5.6 Patrón base para experimentos de *healing*.

5.7. Healing

En el caso de los experimentos de *healing*, se tuvo que desarrollar un conjunto de datos bien particular, con el cual fue posible garantizar el correcto funcionamiento del algoritmo. En la figura 5.18 se indica el patrón de datos utilizado para la experimentación del algoritmo de *healing*. A partir de este, se generó un *layout* más extenso.

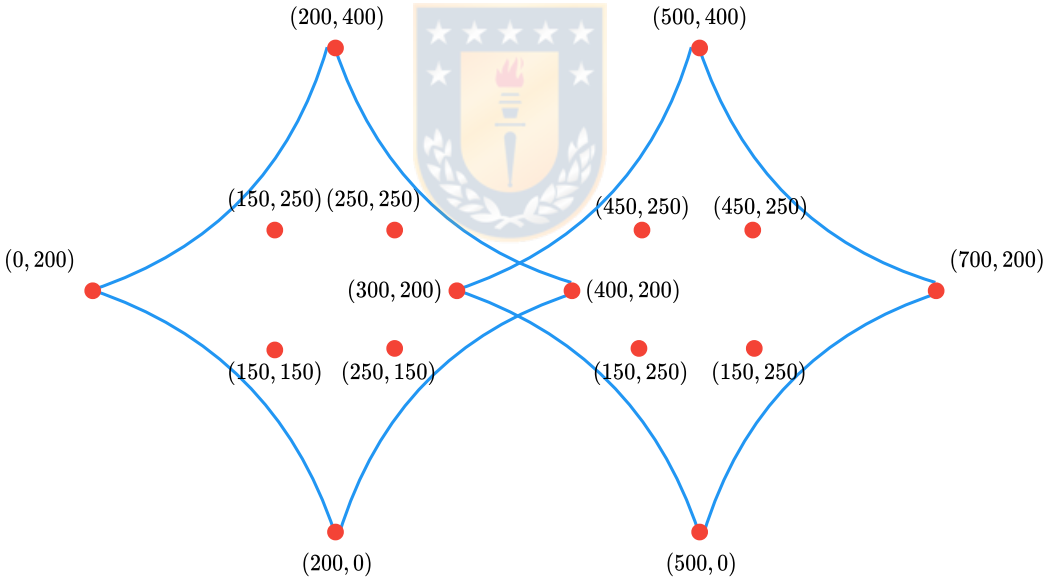


Figura 5.18 Patrón base para experimentos de *healing (Fuente: elaboración propia)*.

El detalle del *layout* utilizado para los experimentos de *healing*, según su número de curvas, segmentos y número de puntos totales es mostrado en la tabla 5.6. Notar que en este caso se trabajó solo con tres representaciones llamadas: beziergons curvas, beziergon segmentos y polígonos. Cada una de estas representaciones son equivalentes a las explicadas en la sección anterior.

5.7.1. Performance

En base a los experimentos, se tiene que la *performance* de las 3 representaciones estudiadas para el algoritmo de healing, se observa que beziergon curvas es 87 más rápido que beziergon segmentos, notar además que todavía requiere bastante trabajo desarrollar una implementación capaz de alcanzar a la versión poligonal, dado que esta utiliza estrategias más eficiente para etapas como el procesamiento de clusters, en cuyo caso los construye en a medida que la *sweepline* los recorre.

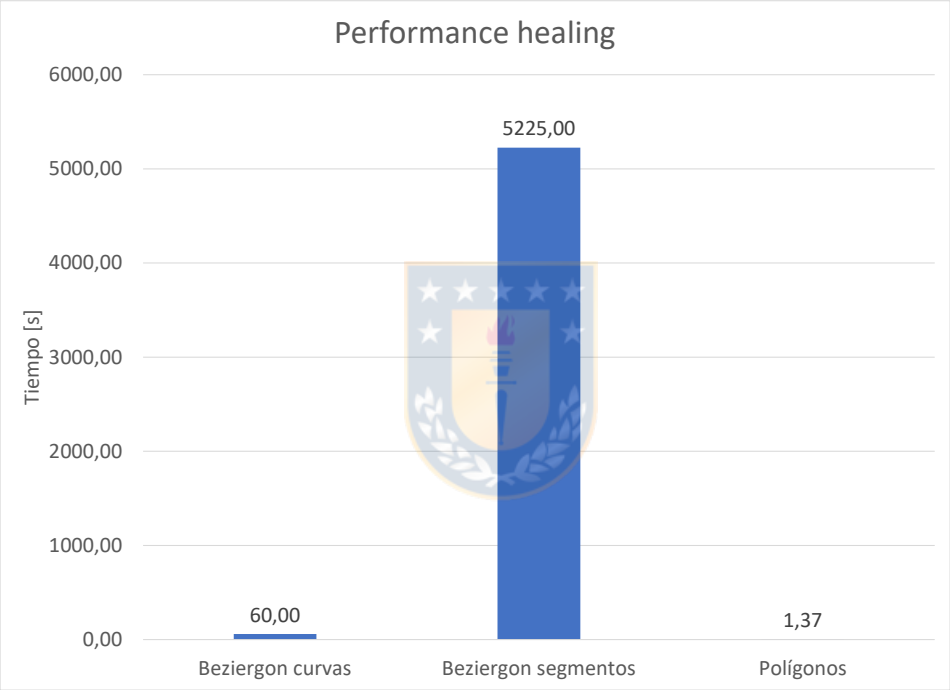
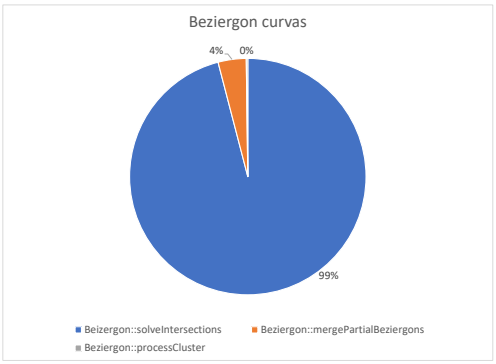


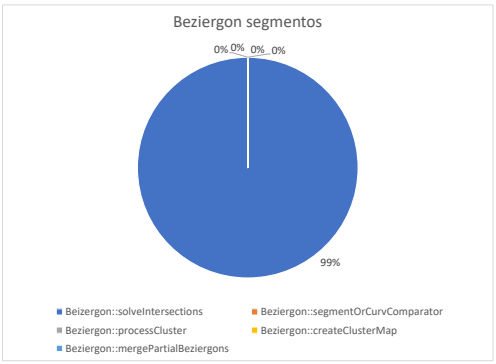
Figura 5.19 *Performance* algoritmo de *healing (Fuente: elaboración propia)*.

5.7.2. Uso de memoria

Por otro lado, respecto al uso de memoria, la representación basada en curvas, también tiene ventajas respecto a la representación basada en segmentos, requiriendo 5.5 veces menos memoria. Se debe considerar que en este experimento muy particular, el uso de memoria de beziergon curvas, se asemeja mucho a los requerimientos de memoria de la representación poligonal, pero cabe recordar



(a) Funciones más costosas algoritmo de *healing*, basado en beziergons patrón ILT *(Fuente: elaboración propia)*.



(b) Funciones más costosas algoritmo de *healing*, basado en beziergons patrón óptico *(Fuente: elaboración propia)*.

Figura 5.20 Funciones más costosas del algoritmo de *healing*, para las representaciones de beziergons, respecto a los patrones ILT y óptico *(Fuente: elaboración propia)*.

que esta última hace uso de estrategias mucho más agresivas para el manejo de clusters, por lo que si se lograra implementar una versión equivalente pero basada en curvas, también podrian aprovecharse esos beneficios.

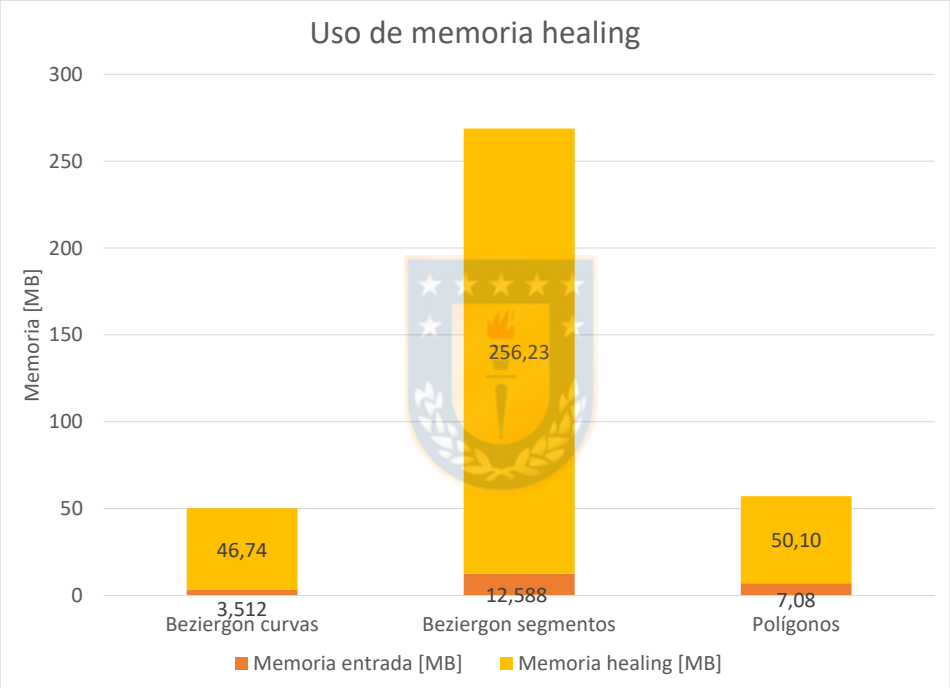


Figura 5.21 Uso de memoria algoritmo de *healing (Fuente: elaboración propia)*

Capítulo 6

Conclusiones

El propósito de este trabajo se centra en evaluar una representación de datos en diseños en VLSI, basada en curvas spline, respecto a la representación clásica basada en polígonos. El objetivo es buscar ventajas haciendo uso de curvas spline, para representar datos dotados altamente de curvas. Las métricas de comparación han sido: el tiempo de ejecución, uso de memoria y calidad de la representación al aplicarse operaciones geométricas sobre la representación.

De los resultados de performance, es posible identificar claras ventajas, cuando se trata de transformaciones lineales aplicadas sobre la representación propuesta respecto a la implementación en su versión poligonal. Por otro lado, en el algoritmo clipping, es posible encontrar ventajas cuando los datos que se buscan representar provienen de geometrías que describen datos de tipo óptico. Sin embargo, no se debe olvidar que al hacer la comparación de la estructura propuesta, respecto a la representación de datos del tipo beziergon segmentos, se pueden evidenciar ventajas en cuanto a performance, tanto para el patrón ILT y óptico.

Otra de las métricas a observar en el comportamiento de la representación propuesta, se requiere al uso de memoria. Existen dos análisis de interese, la primera respecto al uso de memoria para mantener la representación de los datos de entrada, resultó evidente que representar datos mediante curvas de Bézier permitió disminuir drásticamente el uso de memoria. Por otro lado, respecto al uso de memoria necesario para operar con la nueva representación, no se observaron requerimientos excesivos de memoria, se comportó en forma similar a la representación poligonal.

Conclusiones respecto a calidad, deben analizarse con mayor detalle. Por un lado, las curvas de Bézier teóricamente permiten tener una representación con

infinita resolución, sin embargo, esta resultará siempre limitada por procesos de discretización. Si bien en los experimentos de clipping, respecto a calidad, al parecer la representación propuesta, tiende a tener un comportamiento tal que la calidad disminuye, esto se debe a que en los experimentos se consideró los procesos de discretización de curvas a segmentos se harían a una tolerancia igual a la dada para convertir los curvas (curvas dadas en los datos de entrada) a la representación poligonal, contal de hacer una comparación justa, pero no debería haber impedimentos para usar tolerancias que permitan explotar mejor el uso de curvas de Bézier.

El contexto de estudio de la representación propuesta en este trabajo, se focaliza en la etapa de preparación de datos de máscara de un diseño en VLSI, siendo esta una de las últimas etapas dentro de flujo de diseño.así, para emular los datos que recibe esta etapa ha sido necesario la creación de datos artificiales. En un flujo de diseño de producción se debería proveer al resto de herramientas de diseño, una manera de soportar curvas de tipo Bézier, con tal de poder completar el flujo de diseño completo.

Otro resultado de interés, es que puede utilizarse para la especificación de un formato de datos , mediante el cual pueden describirse grandes volúmenes asociados un diseño en VLSI, pero con claras ventajas usando curvas de tipo spline.

Por otro lado, como experiencia del trabajador se puede decir que elaborar aquellos algoritmos que operan sobre curvas se vuelven más complejos debido a que el manejo mediante curvas debe ser realizado abordando un mayor número de condiciones lógicas.

Bibliografía

[1] Barry, P. and Goldman, R. (1988). De casteljau-type subdivision is peculiar to bézier curves. *Computer-aided design*, 20(3):114–116.

[2] Bartolf, H. (2016). *Nanoscale-Precise Coordinate System: Scalable, GDSII-Design*, pages 37–42. Springer Fachmedien Wiesbaden, Wiesbaden.

[3] Chakraborty, A. (2014). An extension of weiler-atherton algorithm to cope with the self-intersecting polygon. *arXiv preprint arXiv:1403.0917*.

[4] Chen, Y., Kahng, A., Robins, G., Zelikovsky, A., and Zheng, Y. (2005). Evaluation of the new oasis format for layout fill compression. pages 377 – 382.

[5] Greiner, G. and Hormann, K. (1998). Efficient clipping of arbitrary polygons. *ACM Transactions on Graphics (TOG)*, 17(2):71–83.

[6] Kahang, A., Lienig, J., Markov, I., and Hu, J. (2011). *VLSI Physical Design: From Graph Partitioning to Timing Closure*. Springer.

[7] Kahng, A. B., Lienig, J., Markov, I. L., and Hu, J. (2011). *VLSI physical design: from graph partitioning to timing closure*. Springer Science & Business Media.

[8] Martínez, F., Rueda, A. J., and Feito, F. R. (2009). A new algorithm for computing boolean operations on polygons. *Computers & Geosciences*, 35(6):1177–1185.

[9] Peng, Y., Yong, J., Zhang, H., and Sun, J. (2005). Efficient algorithm for general polygon clipping.

[10] Puri, S. and Prasad, S. K. (2014). Output-sensitive parallel algorithm for polygon clipping. In *2014 43rd International Conference on Parallel Processing*, pages 241–250.

[11] Schneider, B.-O. (1992). Accelerating polygon clipping. In *Eurographics Workshop on Graphics Hardware*, pages 24–43.

[12] Vatti, B. R. (1992). A generic solution to polygon clipping. *Commun. ACM*, 35(7):56–63.

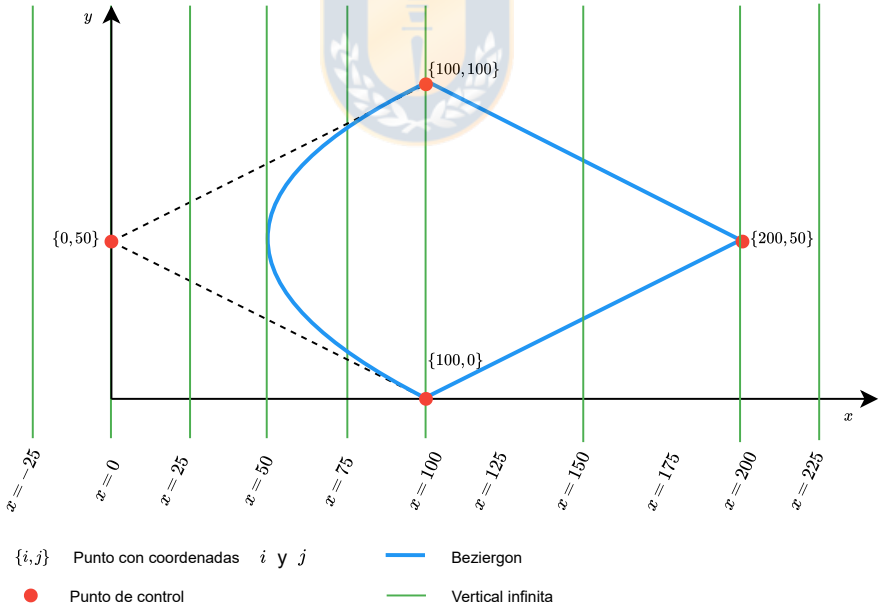
[13] Yeap, K. H. and Nisar, H. (2018). Introductory chapter: Vlsi. In *Very-Large-Scale Integration*. IntechOpen.

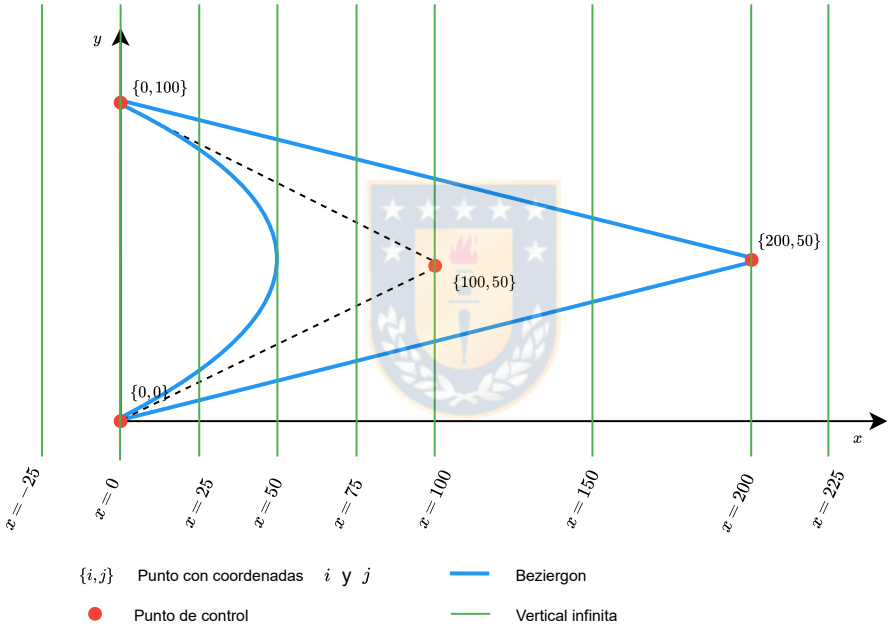
Apéndice A

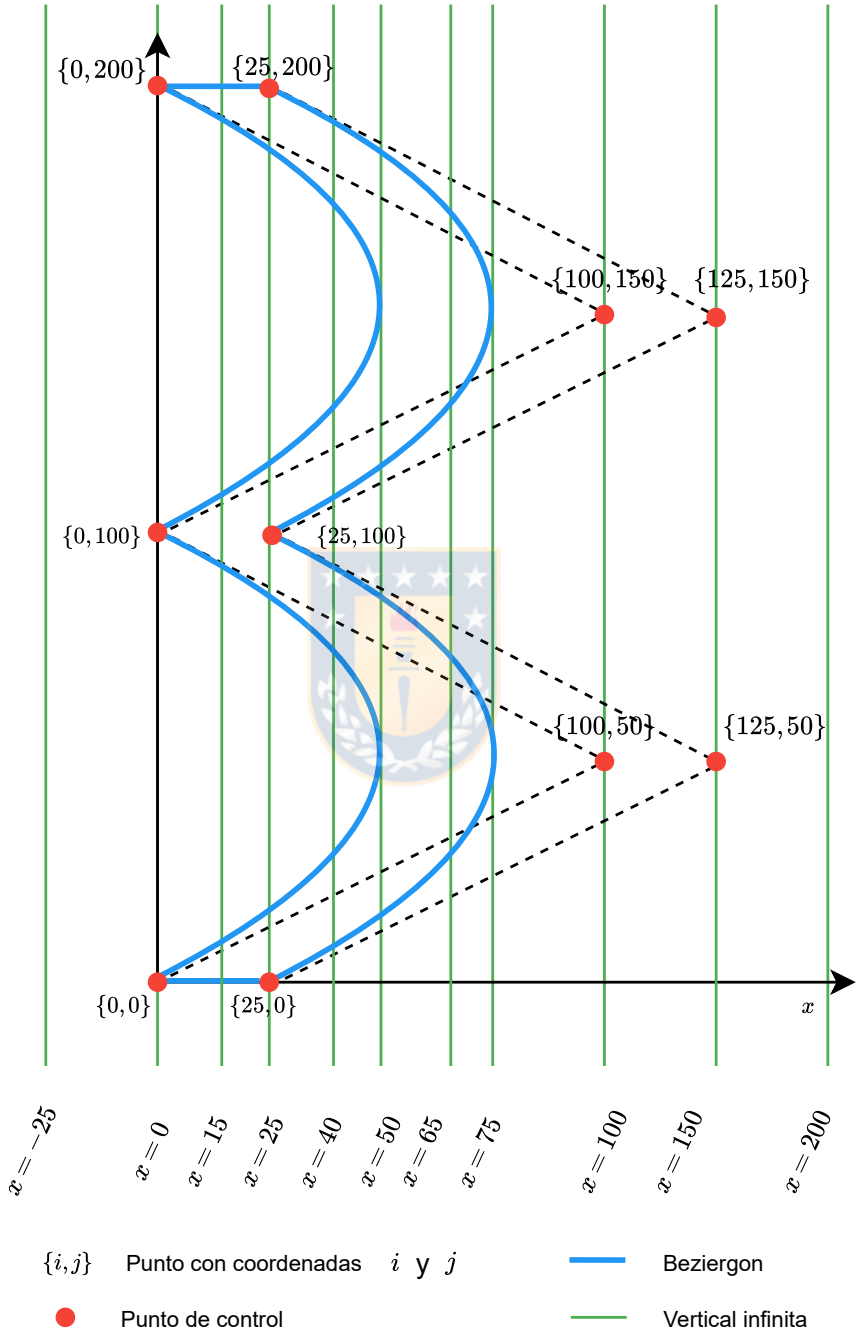
Algoritmos

A.1. Clipping

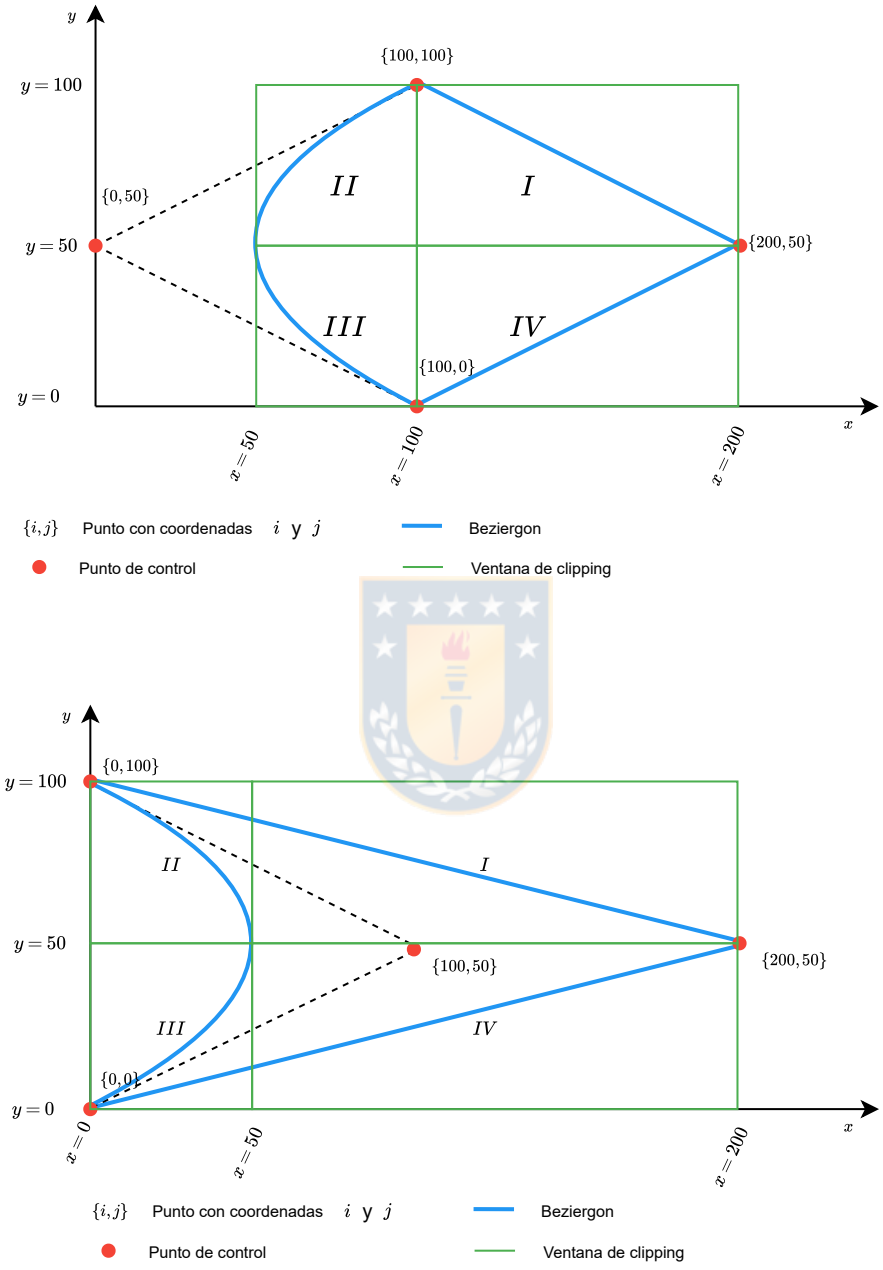
A.1.1. Clipping izquierdo

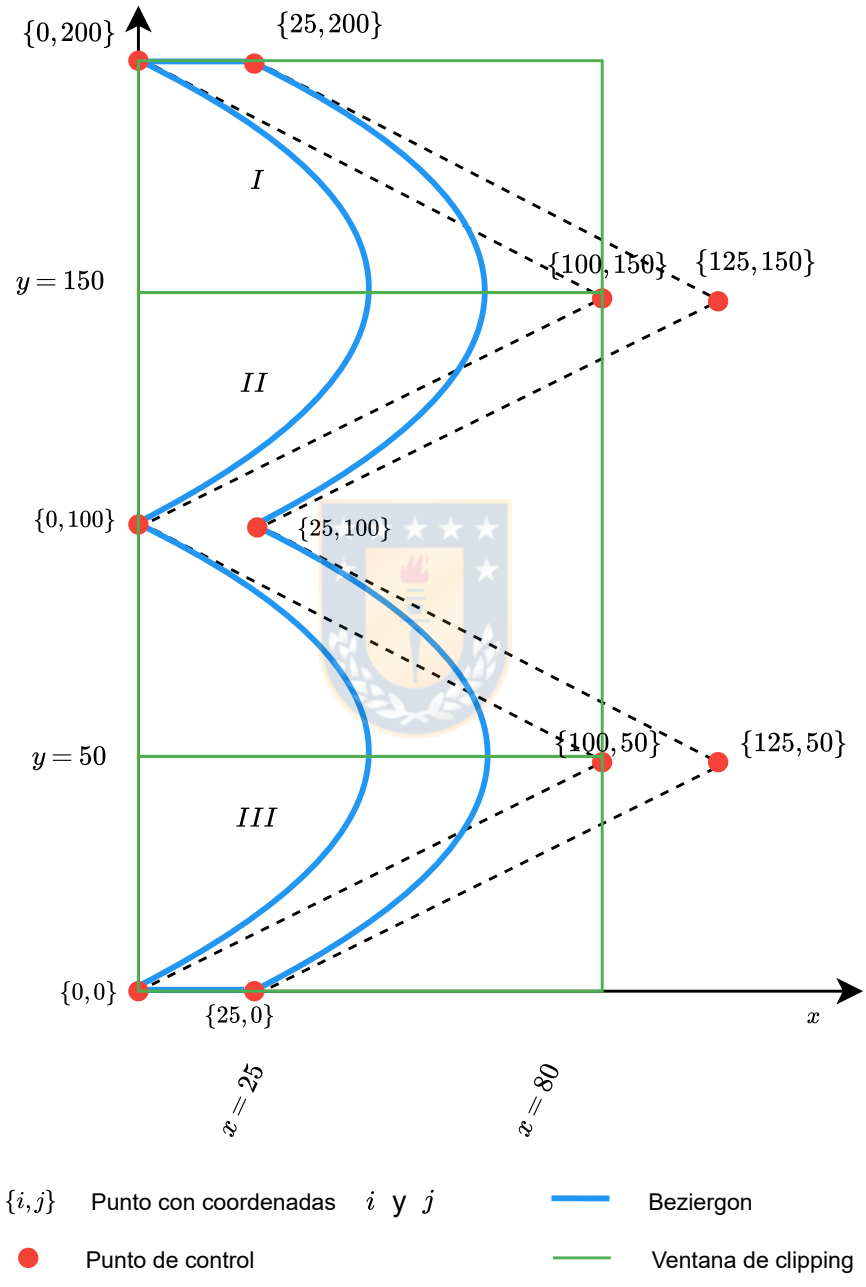



y
{0,100}
{200,50}
{100,50}
{0,0}
x
x = −25
x = 0
x = 25
x = 50
x = 75
x = 100
x = 150
x = 200
x = 225
{i,j} Punto con coordenadas i y j
Punto de control
Beziergon
Vertical infinita

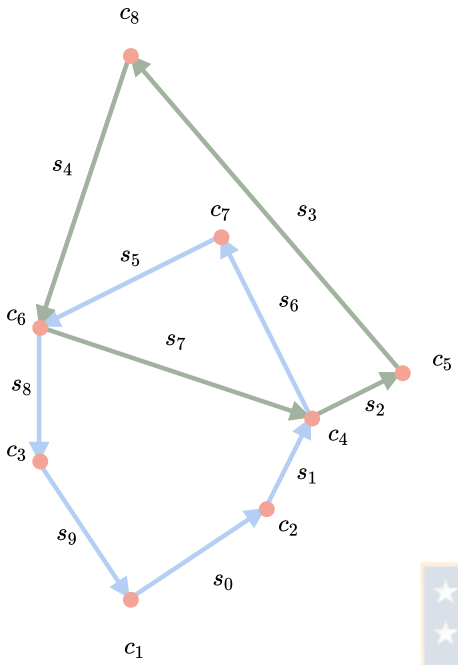

{0,200}
{25,200}
{100,150}
{125,150}
{0,100}
{25,100}
{100,50}
{125,50}
{0,0}
{25,0}
x
x = −25
x = 0
x = 15
x = 25
x = 40
x = 50
x = 65
x = 75
x = 100
x = 150
x = 200
{i,j} Punto con coordenadas i y j
Punto de control
Beziergon
Vertical infinita

A.1.2. Clipping de ventana

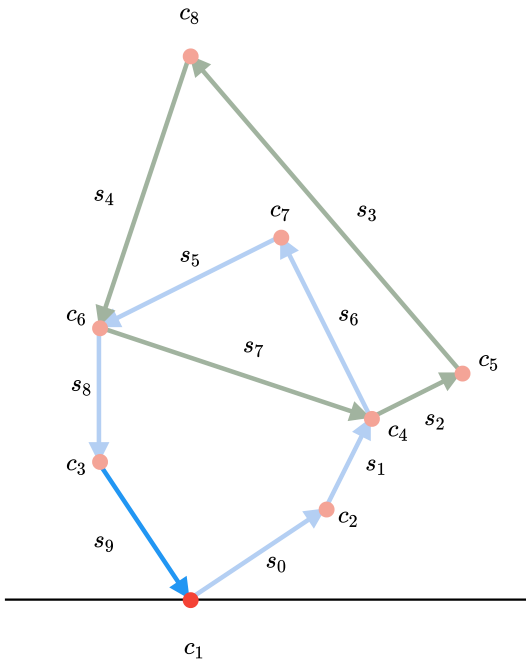



{0,200}
{25,200}
I
y = 150
{100,150}
{125,150}
II
{0,100}
{25,100}
y = 50
{100,50}
{125,50}
III
{0,0}
{25,0}
x
x = 25
x = 80
{i,j} Punto con coordenadas i y j
Beziergon
Punto de control
Ventana de clipping

A.2. Healing evolución lista activa

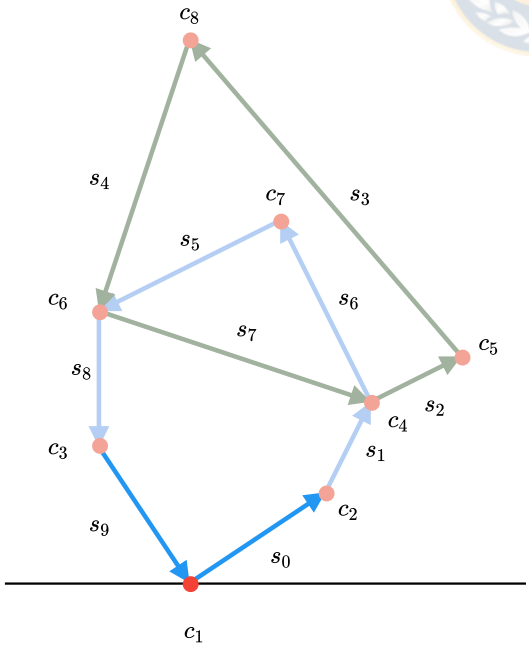


L : $fc = 0$



L : $\boxed{s_9, fc = 1}$ $0 \to 1$ ✓

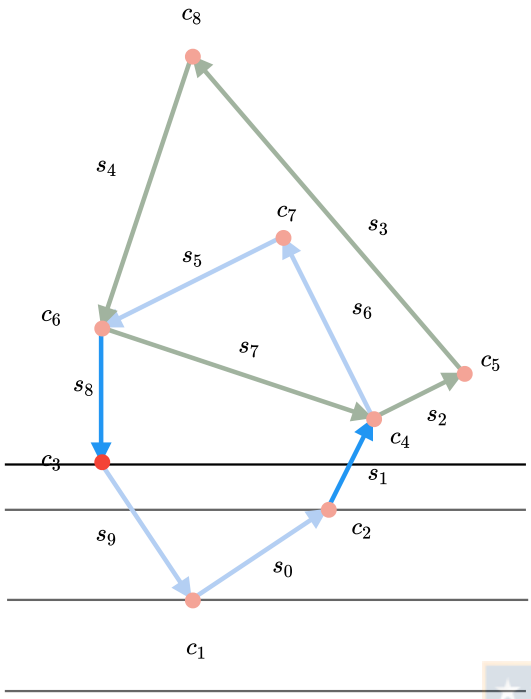
L : $fc = 0$



$1 \to 0$ ✓

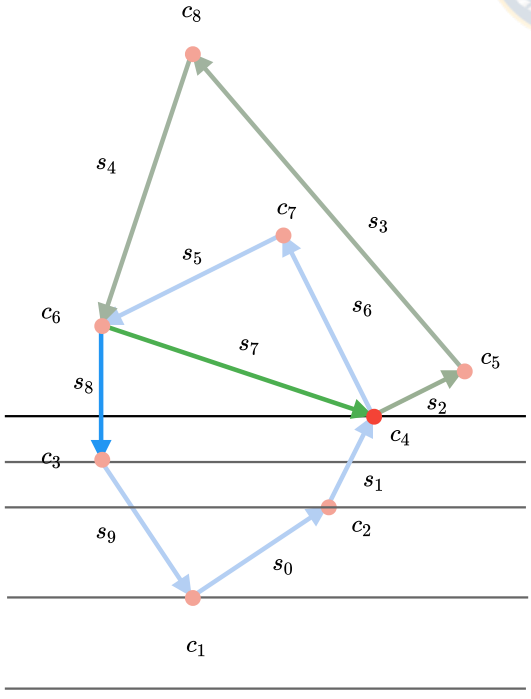
L : $s_9, fc = 1$ | $\boxed{s_0, fc = 0}$

L : $fc = 0$



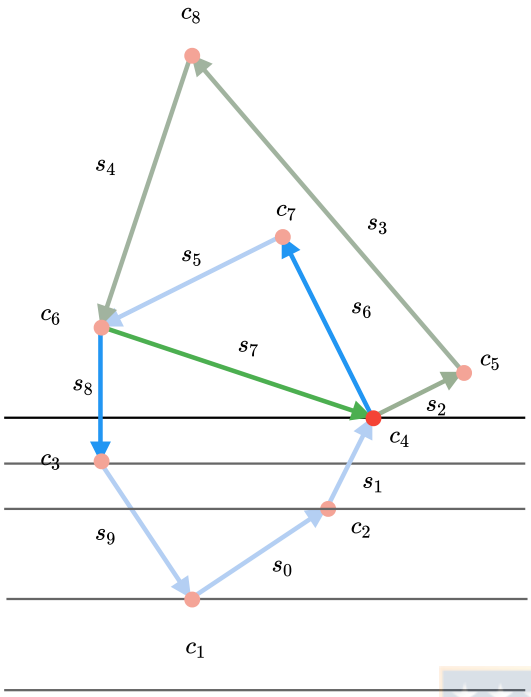
$0 \to 1$ ✔

$L:$	$s_8, fc = 1$	$s_1, fc = 0$
$L:$	$s_9, fc = 1$	$s_1, fc = 0$
$L:$	$s_9, fc = 1$	$s_0, fc = 0$
$L:$	$fc = 0$	



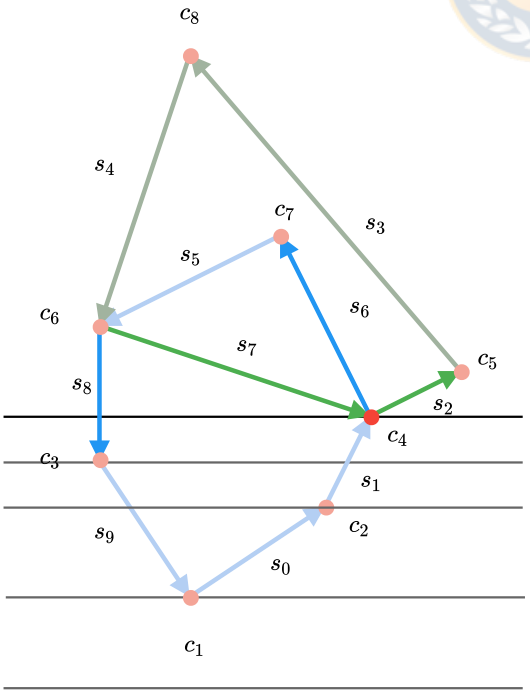
$1 \to 2$ ✘

$L:$	$s_8, fc = 1$	$s_7, fc = 2$
$L:$	$s_8, fc = 1$	$s_1, fc = 0$
$L:$	$s_9, fc = 1$	$s_1, fc = 0$
$L:$	$s_9, fc = 1$	$s_0, fc = 0$
$L:$	$fc = 0$	



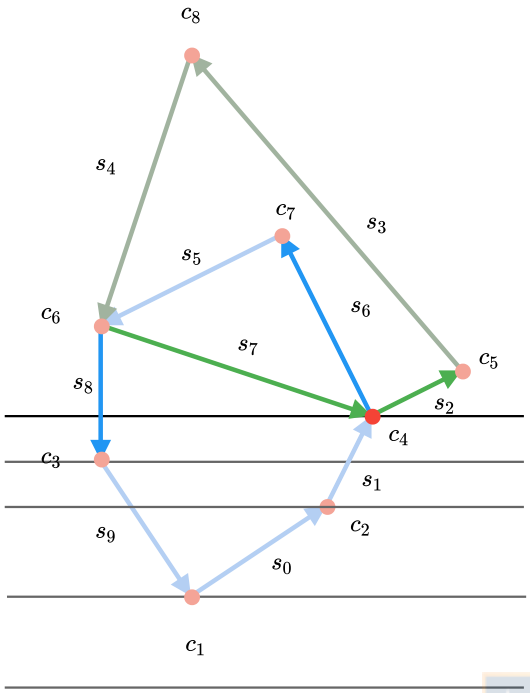
$2 \rightarrow 1$ ✗

$L:$	$s_8, fc=1$	$s_7, fc=2$	$s_6, fc=1$
$L:$	$s_8, fc=1$	$s_1, fc=0$	
$L:$	$s_9, fc=1$	$s_1, fc=0$	
$L:$	$s_9, fc=1$	$s_0, fc=0$	
$L:$	$fc=0$		



$1 \rightarrow 0$ ✓

$L:$	$s_8, fc=1$	$s_7, fc=2$	$s_6, fc=1$	$s_2, fc=0$
$L:$	$s_8, fc=1$	$s_1, fc=0$		
$L:$	$s_9, fc=1$	$s_1, fc=0$		
$L:$	$s_9, fc=1$	$s_0, fc=0$		
$L:$	$fc=0$			



$1 \to 0$ ✓

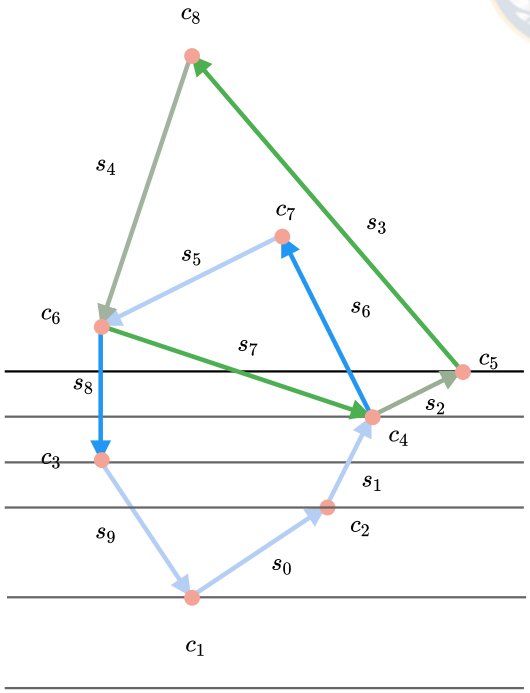
L : $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_2, fc = 0$

L : $s_8, fc = 1$ | $s_1, fc = 0$

L : $s_9, fc = 1$ | $s_1, fc = 0$

L : $s_9, fc = 1$ | $s_0, fc = 0$

L : $fc = 0$



$1 \to 0$ ✓

L : $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$

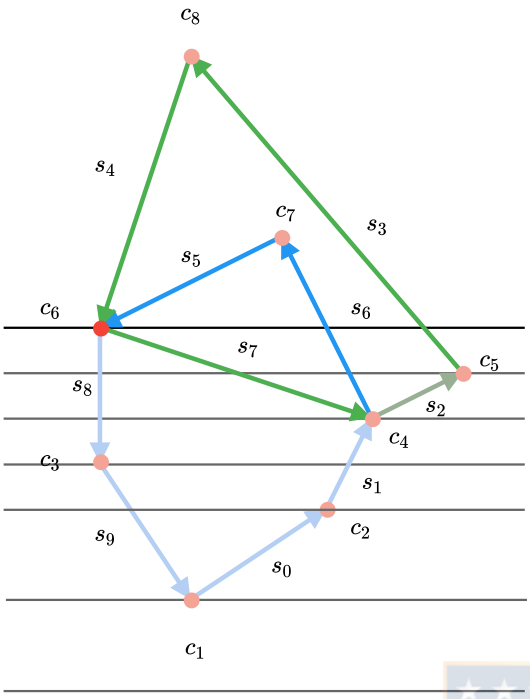
L : $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_2, fc = 0$

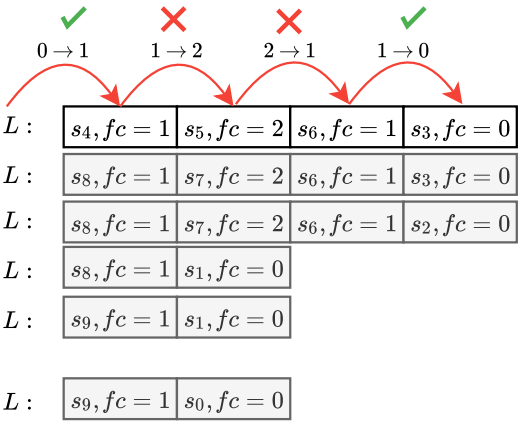
L : $s_8, fc = 1$ | $s_1, fc = 0$

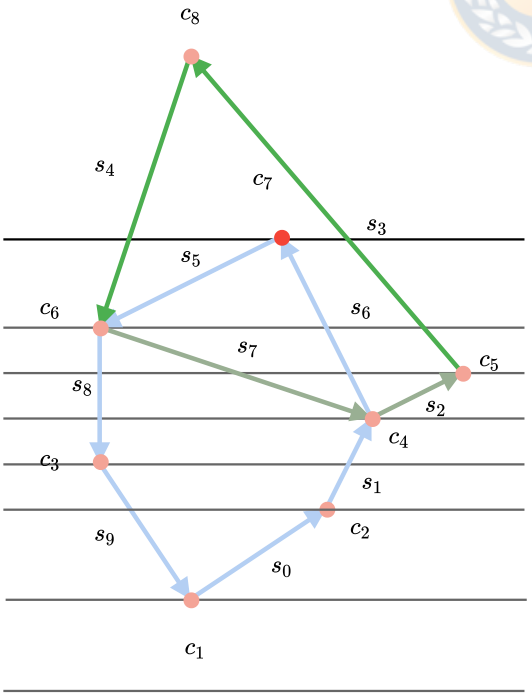
L : $s_9, fc = 1$ | $s_1, fc = 0$

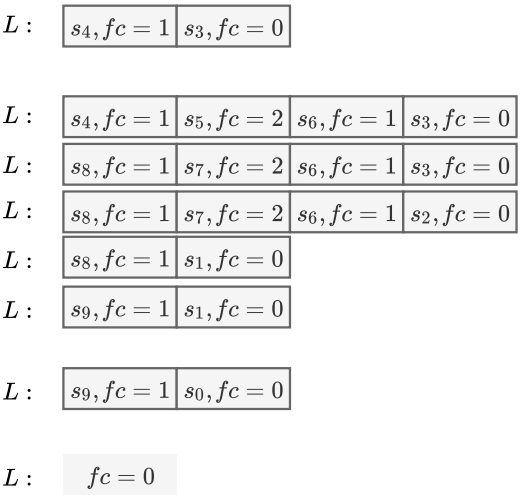
L : $s_9, fc = 1$ | $s_0, fc = 0$

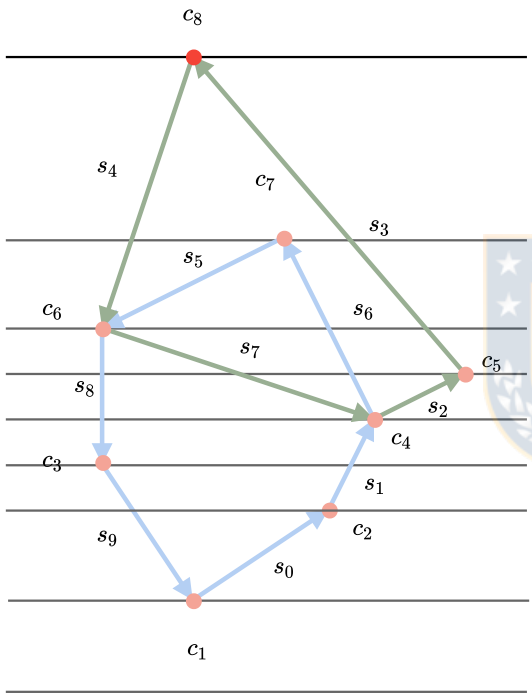
L : $fc = 0$


c_8
s_4
c_7
s_3
s_5
c_6
s_6
s_7
c_5
s_8
s_2
c_4
c_3
s_1
c_2
s_9
s_0
c_1


$0 \to 1$
$1 \to 2$
$2 \to 1$
$1 \to 0$
$L:$ $s_4, fc = 1$ | $s_5, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$
$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$
$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_2, fc = 0$
$L:$ $s_8, fc = 1$ | $s_1, fc = 0$
$L:$ $s_9, fc = 1$ | $s_1, fc = 0$
$L:$ $s_9, fc = 1$ | $s_0, fc = 0$
$L:$ $fc = 0$


c_8
s_4
c_7
s_3
s_5
c_6
s_6
s_7
c_5
s_8
s_2
c_4
c_3
s_1
c_2
s_9
s_0
c_1


$L:$ $s_4, fc = 1$ | $s_3, fc = 0$
$L:$ $s_4, fc = 1$ | $s_5, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$
$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$
$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_2, fc = 0$
$L:$ $s_8, fc = 1$ | $s_1, fc = 0$
$L:$ $s_9, fc = 1$ | $s_1, fc = 0$
$L:$ $s_9, fc = 1$ | $s_0, fc = 0$
$L:$ $fc = 0$



$L:$ $fc = 0$

$L:$ $s_4, fc = 1$ | $s_3, fc = 0$

$L:$ $s_4, fc = 1$ | $s_5, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$

$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_3, fc = 0$

$L:$ $s_8, fc = 1$ | $s_7, fc = 2$ | $s_6, fc = 1$ | $s_2, fc = 0$

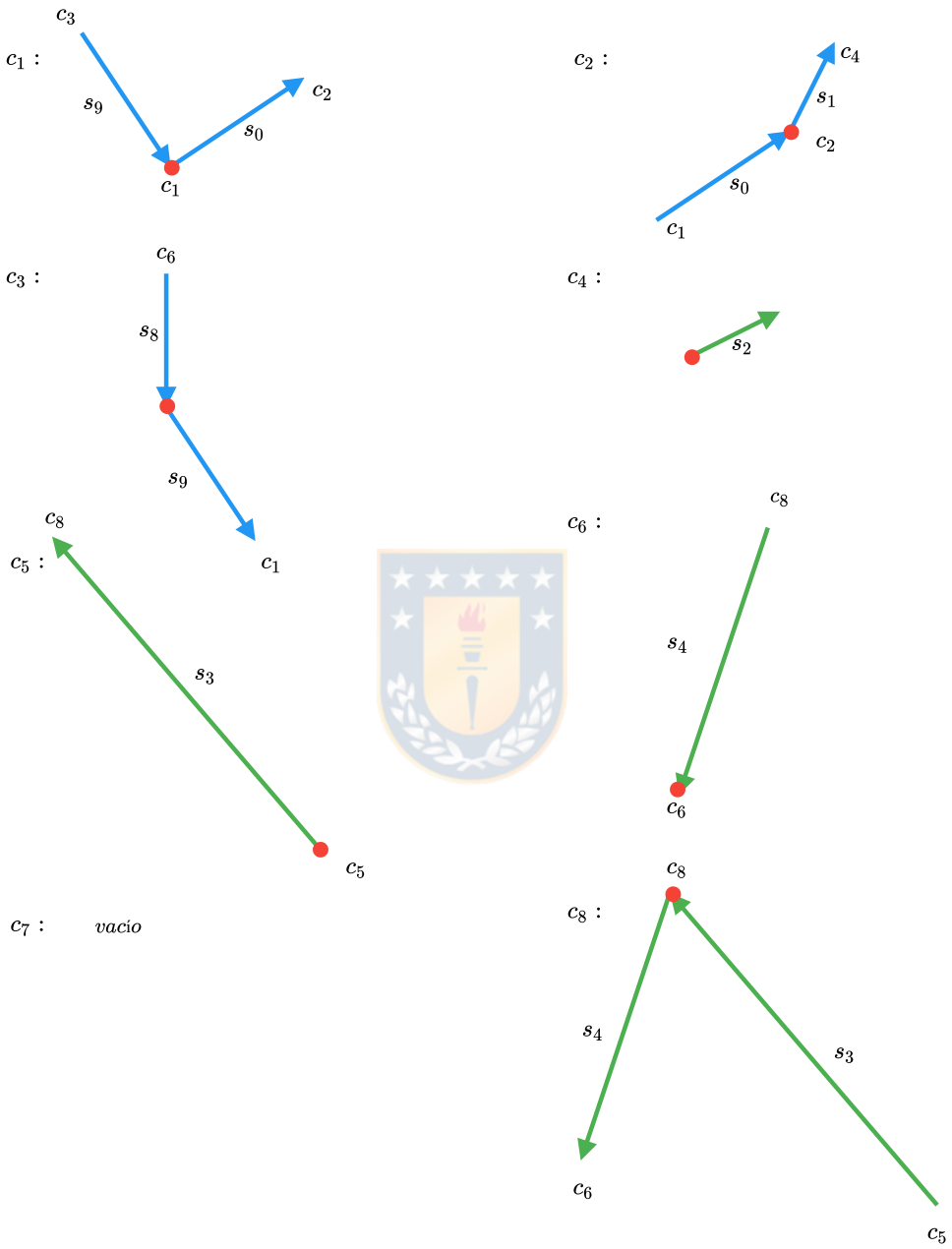
$L:$ $s_8, fc = 1$ | $s_1, fc = 0$

$L:$ $s_9, fc = 1$ | $s_1, fc = 0$

$L:$ $s_9, fc = 1$ | $s_0, fc = 0$

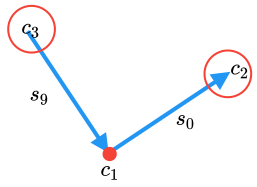
$L:$ $fc = 0$

A.3. Healing procesados

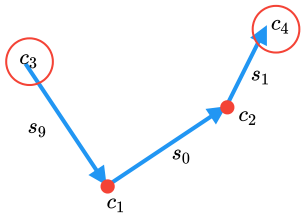


A.4. Healing contornos

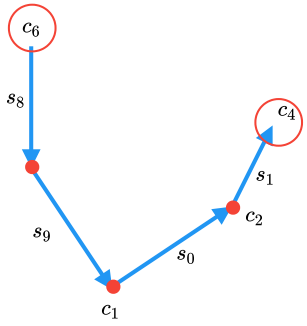
c_1 :



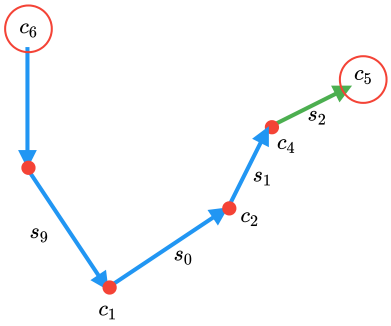
c_2 :



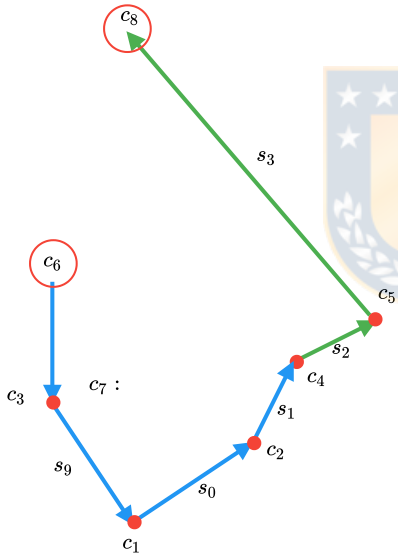
c_3 :



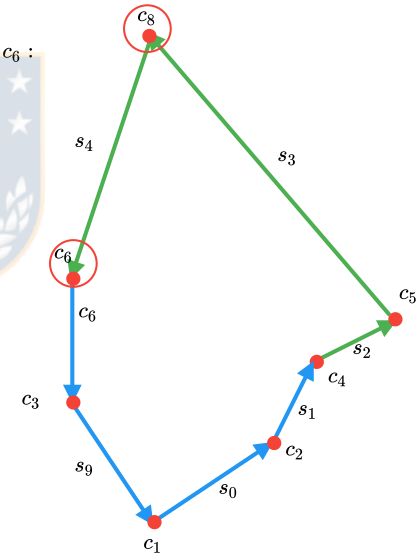
c_4 :



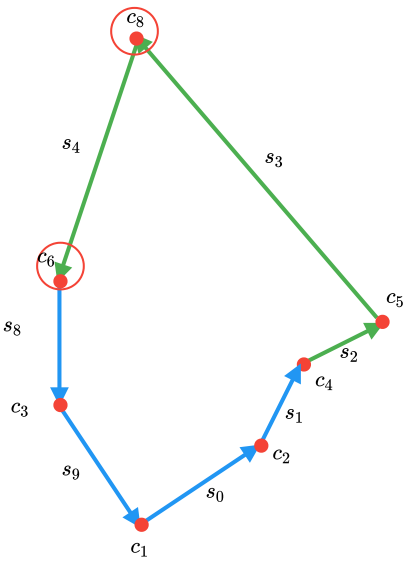
c_5 :



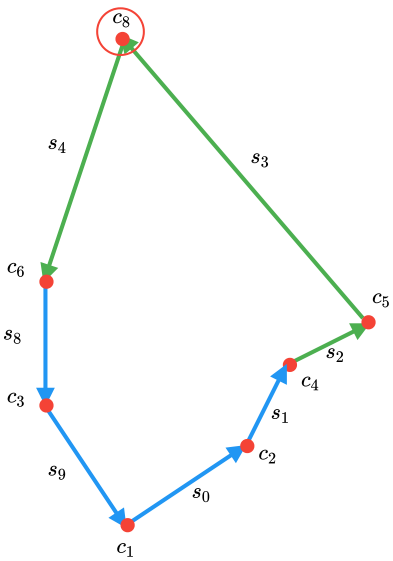
c_6 :



c_7 :



c_8 :



Apéndice B

Datos experimentales

B.1. Datos Artificiales patrón ILT

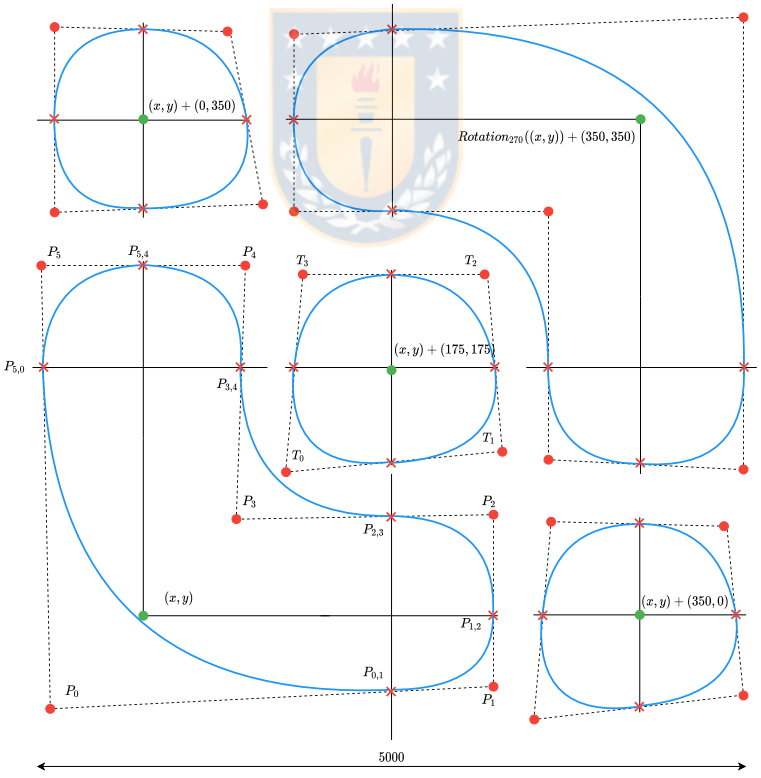


Figura B.1 Datos artificiales basados en un patrón de ILT.

$$P_0 = (x,y) - (rand[1,740], rand[1,740])$$

$$P_1 = (x,y) + (1750,0) + (rand[1,740], -rand[1,740])$$

$$P_2(x,y) + (1750,0) + (rand[1,740], rand[1,740])$$

$$P_3(x,y) + (rand[1,740], rand[1,740])$$

$$P_4 = (x,y) + (0,1750) + (rand[1,740], rand[1,740])$$

$$P_5 = (x,y) + (0,1750) + (-rand[1,740], rand[1,740])$$

$$T_0 = (x,y) + (1750,1750) + (-rand[1,740], -rand(1,740))$$

$$T_1 = (x,y) + (1750,1750) + (rand[1,740], -rand(1,740))$$

$$T_2 = (x,y) + (1750,1750) + (rand[1,740], rand(1,740))$$

$$T_3 = (x,y) + (1750,17500) + (-rand[1,740], -rand(1,740))$$

B.2. Datos Artificiales patrón óptico

$$P_0 = (x,y)$$

$$P_1 = (x,y) + (2500,5000) + (rand(-10,10),0)$$

$$P_2 = (x,y) + (7500,5000) + (rand(-10,10),0)$$

$$P_3 = (x,y) + (10000,0)$$

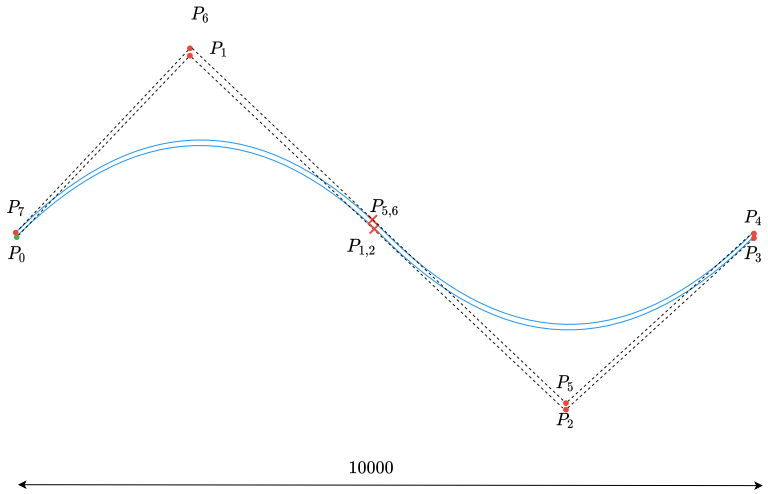


Figura B.2 Datos artificiales que inspirados en diseños de tecnología basados en luz.

$$P_4 = (x,y) + (0,20) + (10000,0)$$

$$P_5 = (x,y) + (0,20) + (7500,5000) + (rand(-10,10),0)$$

$$P_6 = (x,y) + (2500,5000) + (0,20) + (rand(-10,10),0)$$

$$P_7 = (x,y) + (20,0)$$

B.3. Seudocódigo experimentos transformaciones lineales

B.3.1. Performance

B.4. Seudocódigo experimentos clipping

B.4.1. Calidad

Algoritmo 9: Seudocódigo evaluación tiempo de ejecución en beziergons

```
   Input: (Beziergon beziergonsInput)
1  Beziergon beziergonsExpected = beziergonsInput
2  Integer numberOfExperiments = 1000
3  time.start()
4  for numerOfExperiment < numberOfExperiments do
5      Integer variable = random()
6      output = beziergonsInput.transformation(variable)
7  time.stop()
8  print(time)
```

Algoritmo 10: Seudocódigo evaluación tiempo de ejecución en polígonos

```
   Input: (Beziergon beziergonsInput)
1  Float tolerance = 5
2  Polygons polygons= beziergonsInput.getPolygonRepresentation(tolerance)
3  Integer numberOfExperiments = 1000
4  time.start()
5  for numerOfExperiment < numberOfExperiments do
6      Integer variable = random()
7      output = polygons.transformation(variable)
8  time.stop()
9  print(time)
```

Algoritmo 11: Seudocódigo evaluación de calidad en beziergons

```
   Input: (Beziergon beziergonsInput)
1  Beziergons beziergons = beziergonsInput
2  tolerance = 1
3  Polygons polygonsExpected = beziergonsInput.getPolygonRepresentation(tolerance)
4  Integer numberOfExperiments = 1000
5  for numerOfExperiment < numberOfExperiments do
6      Integer variable = random()
7      beziergonsOutput = beziergonsInput.transformation(variable)
8      polygonsOutput = beziergonsOutput.(tolerance)
9      polygonsExpectedOutput = polygonsExpected.transformation(variable)
10     area = Xor(polygonsOutput, polygonsExpectedOutput)
11     print(area)
```

Algoritmo 12: Seudocódigo evaluación de calidad en polígonos

```
   Input: (Beziergon beziergonsInput)
1  Beziergons beziergons = beziergonsInput
2  Float tolerance = 5
3  Polygons polygons = beziergonsInput.getPolygonRepresentation(tolerance)
4  tolerance = 1
5  Polygons polygonsExpected = beziergonsInput.getPolygonRepresentation(tolerance)
6  Integer numberOfExperiments = 1000
7  for numerOfExperiment < numberOfExperiments do
8      Integer variable = random()
9      polygonsOutput = polygons.transformation(variable)
10     polygonsExpectedOutput = polygonsExpected.transformation(variable)
11     area = Xor(polygonsOutput, polygonsExpectedOutput)
12     print(area)
```