ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea in informatica

Grafica Vettoriale e Rasterizzazione

Relatore:
Chiar.mo Prof.
Giulio Casciola

Presentata da:
Giuseppe Balistreri

Sessione III
2017/2018

Alla mia famiglia

Introduzione

In questa tesi ci occuperemo di esplorare il dietro le quinte della "Grafica Vettoriale" o delle così dette immagini vettoriali che tutti indistintamente usiamo, apprezziamo, ma di cui sappiamo molto poco.

Si è scelto di approfondire lo studio di questo argomento perchè è un settore della computer graphics che offre grandi potenzialità e verso cui anche grosse aziende di hardware grafico si stanno orientando.

Nei capitoli che incontreremo, daremo la definizione di "Immagine Vettoriale", vedremo le qualità che le rendono così importanti, daremo uno sguardo ai software esistenti che permettono di gestirle, studieremo gli algoritmi principali per il loro calcolo.

Così facendo capiremo che la parte più complessa e meno nota è come poter rappresentare un disegno vettoriale sui dispositivi raster, ad oggi la tecnologia dominante per i display nei nostri PC, tablet, smarthphone, ecc.

Vedremo quindi varie tecniche di "rasterizzazione" e come migliorare visivamente l'output grafico tramite tecniche di AntiAliasing.

Parte fondamentale del lavoro è stata la realizzazione di una piattaforma per la sperimentazione e messa a punto di numerosi algoritmi di rasterizzazione in modo da poterli confrontare ed analizzare per prestazioni e qualità.

Indice

Introduzione **i**

1 Grafica Vettoriale **1**

1.1 Immagine Vettoriale 2

1.2 Pacchetti Software 5

1.3 Formati Vettoriali 6

1.4 Tablet e Smartphone 7

1.5 Supporto Hardware 7

2 Principi fondamentali **9**

2.1 Curve di Bézier 9

2.1.1 Curva lineare (grado 1) 10

2.1.2 Curva quadratica (grado 2) 10

2.1.3 Curva cubica (grado 3) 11

2.1.4 Disegno di una curva di Bézier 11

2.2 Curve a tratti 12

2.3 Offset di curve 13

2.4 Disegno dell'offset di una curva 14

2.5 Gestione offset nei software 15

2.6 Pixel 16

2.7 Rasterizzazione di linea vettoriale 17

2.8 Riempimento di poligoni 18

2.9 Bitmap 19

3 Antialiasing **21**
3.1 Linea di Xiaolin Wu 22
3.2 SuperSampling 23
3.3 MultiSampling 24
3.4 PostProcessing 27
3.5 Morphological AntiAliasing 27
3.6 Conservative Morphological AntiAliasing 29
3.6.1 Analisi dell'immagine per discontinuità di colore 29
3.6.2 Estrazione dei bordi localmente dominanti 30
3.6.3 Gestione di forme semplici 30
3.6.4 Gestione della forma Z 30

4 Sperimentazione Numerica **33**
4.1 Ambiente di Calcolo 33
4.2 Implementazione degli algoritmi base 34
4.2.1 Bitmap 35
4.2.2 Rasterizzazione di linea Vettoriale 35
4.2.3 Rasterizzazione di Curve di Bezier 36
4.2.4 Riempimento poligoni 38
4.3 MultiSampling (MSAA) 38
4.3.1 Valori di input 39
4.3.2 Supporto al campionamento 40
4.3.3 Triangolazione bordi 41
4.3.4 Riempimento a sinistra (*1) 44
4.3.5 Riempimento a destra (*2) 46
4.3.6 Campionamento 48
4.3.7 Bilanciamento colori 50
4.3.8 Considerazioni finali 50
4.4 Conservative Morphological AntiAliasing (CMAA) 52
4.4.1 Analisi dell'immagine per discontinuità di colore 52
4.4.2 Rilevatore delle forme Z e applicazione del nuovo colore 54
4.4.3 Considerazione finale 56

4.5 Confronto AntiAliasing 57
4.6 Offset di Curve 58
4.7 MSAA nei tracciati 61

Conclusioni **67**

Bibliografia **69**

Elenco delle figure

1.1 Esempio di scala (ridimensionamento), sul lato sinistro troviamo il risultato di un immagine "Raster", sul lato destro si può vedere la sua controparte "Vettoriale". 1
1.2 Processi di conversione 2

2.1 Curva di Bézier (di grado 3) 10
2.2 Suddivisione della curva di Bézier ricorsivamente, i punti verdi sono i punti in cui i tratti superano il test. 12
2.3 Continuità nel seguente ordine (figura A, B,C): G^0, G^1, G^2 . . 12
2.4 Offset della curva 13
2.5 Tracciati nel software Inkscape 15
2.6 Visualizzazione nel dettaglio degli estremi 15
2.7 Graphics Processor Unit 16
2.8 Algoritmo di linea (Jack Elton Bresenham) 17
2.9 Algoritmi di riempimento di poligoni: ScanLine, FloodFill . . 18
2.10 Esempio di bitmap 19

3.1 A sinistra si nota una linea ideale e a destra una linea dopo il processo di Rasterizzazione 23
3.2 A sinistra possiamo trovare Bresenham che arrotonda solamente al pixel vicino, a destra Xiaolin Wu che determina il colore tramite la distanza dal centro del pixel e la linea 23
3.3 Rasterizzazione di linea con antialiasing di Xiaolin Wu's . . . 23

3.4 SSAA, il disegno a risoluzione maggiore viene processato nella sua interezza con il processo di "downsampling" e la media dei pixel extra sarà il colore del pixel nel dispositivo di output. . . 24
3.5 MSAA, campionamento eseguito solo sui bordi, le aree rosse. 25
3.6 Esempi MSAA e Punti campione per MSAA 4x 25
3.7 Esempio di copertura dei sotto-campioni (MSAA) 26
3.8 Riconoscimento forme MLAA, sulla sinistra troviamo i pattern riconosciuti (Z, L, U), sulla destra abbiamo applicato il filtro . 28
3.9 Calcolo area MLAA, come si può notare la seguente forma viene divisa in mezzo e viene gestita come area di trapezoidi . 28
3.10 CMAA, esempi delle forme riconosciute. 29
3.11 Gestione delle forme a 2, 3, 4 bordi 30
3.12 Forma Z nel CMAA, sulla sinistra Gestione della forma Z espandibile, sulla destra la determinazione dell'area occupata dalla forma Z 31

4.1 Rasterizzazione di linee, a sinistra Bresenham, nel centro Xiaolin Wu e a destra Gupta Sproull 36
4.2 Editor Bezier Path 37
4.3 MSAA 16x 39
4.4 Sulla sinistra troviamo una definizione di linee nel verso antiorario, e sulla destra nel verso orario 40
4.5 Esempio di triangolazione dei bordi interni ed esterni 41
4.6 Rappresentazione della funzione LEFTorRIGHT 42
4.7 Triangolazione dei bordi verso sinistra con riempimento a sinistra (angolo acuto) 45
4.8 Triangolazione dei bordi verso sinistra con riempimento a sinistra (angolo normale) 46
4.9 Triangolazione dei bordi verso sinistra con riempimento a destra (angolo acuto) 47
4.10 Triangolazione dei bordi verso sinistra con riempimento a destra (angolo normale) 47

4.11 Nella figura possiamo vedere l'applicazione dell'MSAA solo ai bordi, verificando la copertura dei punti campioni sulla serie di triangoli creati a risoluzione maggiore (di colore viola) . . . 48
4.12 Punti campione per MSAA 4x 49
4.13 Esempio di copertura dei sotto-campioni (MSAA) 50
4.14 MSAA 16x 51
4.15 Rilevamento bordi per pixel 52
4.16 Ricerca Z trovata a Nord-Est 54
4.17 Ricerca Z trovata a Sud-Est 55
4.18 CMAA: conservative morphological antialiasing 56
4.19 Confronto AntiAliasing 57
4.20 Confronto AntiAliasing 57
4.21 AutoIntersezione dell'offset di una curva. 59
4.22 Tracciati senza gestire le connessioni tra le curve. 59
4.23 L'offset sugli spigoli. 60
4.24 Spigolo Vivo 60
4.25 Spigoli: sulla sinistra lo "Spigolo Tagliato", al centro " Spigolo Arrotondato", a destra lo "Spigolo Vivo". 61
4.26 Tracciati con MSAA 16x 62
4.27 Un esempio della triangolazione del tracciato a risoluzione maggiore (viola) 64
4.28 Tracciato completo con Antialiasing 65
4.29 Utilizzo del tracciato per rasterizzare la polilinea della curva con antialiasing. 66

Capitolo 1

Grafica Vettoriale

Figura 1.1: Esempio di scala (ridimensionamento), sul lato sinistro troviamo il risultato di un immagine "Raster", sul lato destro si può vedere la sua controparte "Vettoriale".

Prima di poter parlare di grafica vettoriale, che può essere immaginato come un ramo del più generale argomento che è la **" Grafica al computer"**, (in inglese "Computer Graphichs" indicata spesso con CG o CGI), è bene definire quella disciplina informatica che riguarda la creazione e manipolazione di immagini, foto e filmati realizzate tramite un computer.

I primi passi fatti da questa disciplina risalgono agli inizi degli anni 50 quando cresceva il bisogno di controllare le macchine tramite i computer (soprattutto per la produzione industriale o per scopi militari). Agli inizi la Computer Graphics fu una disciplina accessibili a pochi, visto l'elevato costo

delle macchine di calcolo e dei terminali grafici, ma con l'avvento dei personal computer e delle schede grafiche chiunque ha potuto avvicinarvisi e la sua diffusione è stata vastissima.

1.1 Immagine Vettoriale

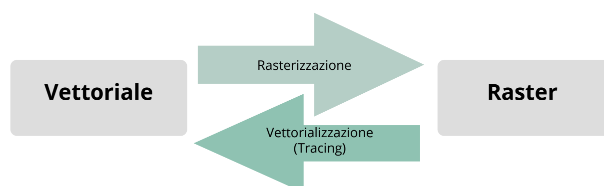

Figura 1.2: Processi di conversione

Nella grafica al computer ci sono due modi per poter definire un'immagine: quella **Raster** che è una matrice di valori di intensità associata alla matrice dei pixel e quella **Vettoriale** che è la descrizione matematica di primitive geometriche come punti, segmenti di retta, archi, curve di Bézier che separano regioni di colore differente. Le prime sono realizzate tramite software appositamente basati sul pixel o acquisite tramite scanner e fotocamere, esse hanno un numero di pixel fissi e vengono di solito usate per rappresentare foto (immagini di vita reale) o per simulare colori di un materiale (come le texture); le seconde vengono create tramite dei software basati sulla grafica vettoriale e sono solitamente usate per creare loghi, font, immagini di pubblicità, illustrazioni.

Bisogna sapere che per entrambe le modalità solitamente l'output di destinazione è sempre un dispositivo raster, cioè sia che approdi su una stampante o su uno schermo, prima di arrivare a destinazione ha bisogno di un passaggio fondamentale chiamato "Rasterizzazione", quindi anche un immagine vettoriale alla fine dovrà essere convertita in un equivalente immagine raster per poter essere visualizzata o stampata.

Una delle proprietà più importante di un'immagine Vector, detta anche a risoluzione infinita è la scalabilità, cioè un ridimensionamento (zoom) senza perdita di qualità, perché le sue funzioni non sono in alcun modo collegate alla risoluzione (cosa che non avviene nella modalità Raster). È possibile esprimere i dati in un linguaggio più vicino all'essere umano (riusciamo a capirne il contenuto anche prima della Rasterizzazione), diversamente da un'immagine Raster (è difficile intuire il contenuto di un'intera immagine raster dalla matrice dei valori di intensità), inoltre un'immagine vettoriale occupa meno spazio di una raster. La grafica vettoriale ha un notevole utilizzo in molteplici campi: nell'architettura, nell'editoria, nell'ingegneria e nella grafica realizzata al computer. Un evidente esempio sono i font sui nostri personal computer, infatti quasi tutti sono realizzati in modo vettoriale così da poter essere scalati senza perdita di qualità.

Segue una breve sintesi delle proprietà [1] che contraddistinguono nel bene e nel male le due modalità:

RASTER

- Basato sul pixel, vincolato alla risoluzione.
- I software relativi sono ottimi per editare foto, in generale per immagini che hanno una certa continuità di colori e trasparenze molto dolci.
- Non si ha una scala ottimale, l'immagine risulterà sgranata.
- È difficile convertire un immagine Raster in una Vector.
- I più comuni software di editing sono: Photoshop, Gimp (free).

VECTOR

- Basato su delle funzioni matematiche, non vincolato alla risoluzione.
- I software relativi sono ottimi per la creazione di un logo, per illustrazioni, disegni tecnici, in generale per rappresentare forme nette che possiamo esprimere tramite una funzione.
- Può essere scalato senza perdità di qualità, stampe sempre nitide, forme sempre perfette.
- È facile convertire un'immagine Vector in una raster.
- I più comuni software di editing sono: Illustrator, CorelDraw, Inkscape (free).

1.2 Pacchetti Software

Esistono diversi pacchetti software sia liberi che a pagamento che permettono di realizzare immagini vettoriali, i più famosi sono:

- **"Inkscape"**: è un software libero multipiattaforma, nato nel 2003 come fork del programma "Sodipodi" a seguito di divergenze sul possibile sviluppo futuro, passò dal C al C++ e molti miglioramenti furono fatti negli anni anche all'interfaccia grafica a tal punto da renderlo competitivo con software commerciali. Permette l'utilizzo di strumenti molto potenti per la creazione e manipolazione dei disegni vettoriali che aumentano di versione in versione, offre piena compatibilità ai formati standard (XML/SVG e CSS) ed è compatibile anche con formati proprietari di Adobe Illustrator, Macromedia FreeHand, Corel Draw.

- **"Adobe Illustrator"** è un software commerciale disponibile per i Sistemi Operativi MacOs e Microsoft Windows scritto in C++, la sua prima pubblicazione fu nel lontano 1987 per Apple Macintosh, oltre ad essere venduto singolarmente viene reso disponibile nella Adobe Creative Suite.

Esistono anche dei linguaggi che permettono di realizzare grafica vettoriale tramite semplice testo, di solito hanno bisogno di essere interpretati ed uno dei più famosi è **"PGF/TikZ "**, sono una coppia di linguaggi per produrre grafica vettoriale tramite espressioni geometriche e algebriche. PGF è un linguaggio di basso livello mentre Tikz è una serie di macro di alto livello che possono essere utilizzate all'interno del popolare LaTex.

Altro progetto molto interessante basato sulla grafica vettoriale è **"Cairo"**, una libreria grafica 2d con il supporto per più dispositivi di output, supporta attualmente X Window System (tramite Xlib e XCB), Quartz, Win32, buffer di immagini, PostScript, PDF e output di file SVG. Cairo fornisce operazioni simili agli operatori di disegno, come PostScript e il PDF, è implementato e scritto in C oltre ad essere un software gratuito è disponibile

per essere ridistribuito e/o modificato in base alla licenza GNU Lesser General Public License (LGPL) versione 2.1 o Mozilla Public License (MPL) versione 1.1. Cairo è usato in molti progetti di rilievo, ad esempio l'engine grafico del layout di Mozilla Firefox fa uso di cairo per disegnare contenuti svg e canvas, anche la libreria plopper ne fa uso per visualizzare pdf, pure Inkscape ne fa uso in una sua feature.

1.3 Formati Vettoriali

Abbiamo accennato in precedenza che il disegno vettoriale è scalabile senza perdità di qualità, perché il disegno finale viene generato alla massima risoluzione disponibile e se viene effettuata una trasformazione (ad esempio una scala) lo stesso viene rigenerato con la sua massima espressione possibile. Grazie a queste proprietà nel mondo del World Wide Web, verso la fine degli anni 90 si è sentito il bisogno di stabilire un formato standard che permetta di inserire un file vettoriale all'interno del codice delle pagine, senza aggiunta di plugin (o altre estensioni).

Il **Word Wide Web Consortium** (detta anche **W3C**, è un organizzazione non governativa Internazionale con il principale scopo di sviluppare le potenzialità del WWW) sollecitava i vari produttori a presentare proposte per un formato Standard decidendo poi di sviluppare il formato **SVG (Scalable Vector Graphics)**, che diverrà lo standard per tutti i Browser, si tratta di un linguaggio derivato dall'XML che si pone come obbiettivo di descrivere figure bidimensionali statiche o animate.

Abbiamo anche altri formati di file per la rappresentazione di un'immagine vettoriale, tra cui i più comuni sono AI,CDR, EPS, PDF.

Alcuni esempi di utilizzo di immagini vettoriali:

- **Font:** è un tipo di carattere Vettoriale, dove i caratteri sono definiti da una serie di istruzioni grafiche (PostScript, TrueType e OpenType), sfruttando tutte le caratteristiche del disegno vettoriale un font può essere scalato in qualsiasi direzione, inoltre da un solo stile è possibi-

le ricavare tutti i vari stili (grassetto e corsivo) occupando così meno spazio.

- **Logo:** per creare un logo viene solitamente usata la grafica vettoriale così da poter generare il disegno ed adattarsi al meglio a qualsiasi risoluzione, invece che avere tante immagini per le diverse risoluzioni necessarie.

1.4 Tablet e Smartphone

Anche nei moderni tablet e smartphone è comune usare dove possibile le immagini vettoriali, al fine semplificare il lavoro di uno sviluppatore, anche se le SVG non sono supportate nativamente esistono delle librerie che permettono di leggere questi formati (queste librerie non supportano tutte le funzionalità dell'SVG). Ad esempio Android 5.0 (api 21) fu la prima versione a supportare il disegno vettoriale con **VectorDrawable** e **AnimatedVectorDrawable**, con tool inclusi nell'IDE di sviluppo per importare e convertire dal formato SVG in DrawableVector (anche quì il supporto non comprendeva tutte le funzionalità offerte dall'SVG).

1.5 Supporto Hardware

Per quanto rigurda il supporto Hardware alla grafica vettoriale possiamo vedere come la Nvidia Corporation ha spinto verso quella direzione, infatti esiste un'estensione OpenGL per utilizzare l'accellerazione della GPU per il disegno di path (**NV Path Rendering**, [2]), supportato da schede video **NVIDIA** che montano il processore **CUDA** (un'architettura Hardware per l'elaborazione parallela). Ha un approcio di disegno indipendente dalla risoluzione, occlusione e trasparenze dipendono dall'ordine di disegno, contiene le primitive base per riempimento ed inspessimento di percorsi (come ad esempio moveto, lineto, curveto, arcto, closepath). È in linea con gli standard: PostScript, PDF, TrueType fonts, Flash, Scalable Vector Graphics

(SVG), HTML5 Canvas, Silverlight, Office drawings e si integra con le APIs di Apple Quartz 2D, Khronos OpenVG, Microsoft Direct2D, Cairo, Skia, Qt::QPainter, Anti-grain Graphics.

Spulciando tra gli ultimi studi svolti dalla Nvidia troviamo un brevetto interessante intitolato **"Infinite Resolution Textures"**(Jun, 7, 2018 [3]) che indica come la Nvidia stia investendo fondi per la ricerca indirizzata proprio nell'ambito della grafica vettoriale.

Capitolo 2

Principi fondamentali

I componenti fondamentali alla base della grafica vettoriale sono come accennato in precedenza le linee, gli archi, le curve, lo spessore delle curve (Stroking) e il riempimento di aree (Filling), tutte le varie parti delle immagini sono separabili e solitamente ogni elemento può essere definito separatamente.

Descriveremo in questa sezione i principi fondamentali che fanno parte della grafica vettoriale, ma dando anche uno sguardo agli algoritmi di disegno/conversione raster [4], che devono necessariamente fare parte del nostro studio per il motivo accennato nel capitolo precedente: *"Qualsiasi dispositivo di destinazione sia uno schermo o una stampante sono dispositivi Raster"*.

2.1 Curve di Bézier

La matematica alla base della grafica vettoriale sono le curve di Bézier, introdotte nel 1962 dall'ingegnere francese **Pierre Bézier**, furono usate per progettare le carrozzerie delle automobili (alla Renault). Contemporaneamente usate da **Paul de Casteljau** nel 1959, che le definì con quello che è oggi conosciuto come l'algoritmo di de Casteljau (era nel gruppo scientifico della Citroèn). Una curva di Bézier è una curva parametrica che permette di definire disegni estremamente precisi tramite un poligono di controllo nel-

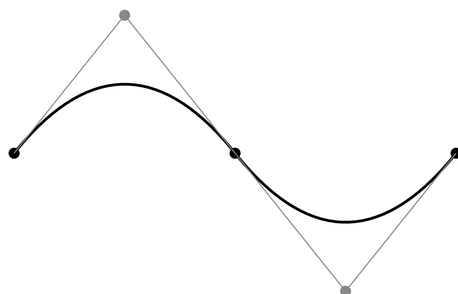

Figura 2.1: Curva di Bézier (di grado 3)

l'intervallo $I = [0, 1]$, inoltre il grado della curva è determinato dal valore $K = n - 1$, con n uguale al numero di vertici del poligono di controllo. Una curva di Bézier è contenuta nell'insieme convesso dei suoi Control Point e le semplici trasformazioni come rotazione, scala e traslazione possono essere applicate ai soli punti di controllo. Tra le più importanti abbiamo le curve lineari, quadratiche e cubiche mentre le curve di grado più alto sono più costose da valutare.

2.1.1 Curva lineare (grado 1)

Corrisponde ad un tratto di retta, presi come estremi P0 e P1, ed ha espressione:

$$C(t) = P0 + t(P1 - P0) = (1 - t)P0 + (t)P1$$

2.1.2 Curva quadratica (grado 2)

Una curva di Bézier quadratica corrisponde ad un arco di parabola, con estremi P0 eP2. Il poligono di controllo è composto da tre punti, e la sua curva sarà contenuta nell'insieme convesso dei suoi control point, e la sua espressione è:

$$C(t) = (1 - t)^2 P0 + 2(1 - t)tP1 + (t^2)P2$$

2.1.3 Curva cubica (grado 3)

Una curva Cubica, ha origine in P0 si dirige verso P1 e arriva in P3 dalla direzione di P2. La curva risiede sempre nella poligonale convessa e i punti della curva non passano per i control point P1 e P2, sono solo necessari per la direzione e la distanza tra P0 e P1 indica quanto la curva si muove verso P2 prima di andare in P3, la sua espressione è:

$$C(t) = (1-t)^3 P0 + 3(1-t)^2 (t) P1 + 3(1-t)^2 (t^2) P2 + (t^3) P3$$

2.1.4 Disegno di una curva di Bézier

Per disegnare una curva di Bézier (sia di grado 1, 2, 3 o più) si prosegue con la realizzazione della polilinea che meglio approssima i punti della curva. Un passo fondamentale per il calcolo dei punti che formano la polilinea è la scelta del parametro $t = [0, 1]$. Una possibile sequenza di facile intuizione, ad esempio una serie di valori equidistanti come $t = [0, 0.1, 0.2, ..., 0.9, 1]$ potrebbe influire negativamente alla rappresentazione della curva ritornando una serie di punti che non sono abbastanza per rappresentarla al meglio, anche aumentandone il numero di elementi non si ottiene la certezza che sia una serie ottimale, inoltre il numero dei punti può essere elevato per alcuni tratti ma insufficiente per altri. La soluzione migliore per il calcolo della polilinea di una curva di Bézier è quella di utilizzare una funzione ricorsiva richiamando se stessa per il tratto sinistro $(t/2)$ ed il rimanente destro, continuando le sue chiamate finchè uno dei tratti della curva presa in esame non sia rappresentabile da un segmento di una retta.

Il nostro algoritmo inserisce il punto iniziale e prosegue con la suddivisione fino alla chiamata ricorsiva in cui il test è verificato, aggiungendo la fine di quel tratto e scorrendo tutta la curva. La funzione di test può essere fatta sul singolo tratto della curva presa in considerazione, testando la distanza in altezza della curvatura e se ≥ 0.5 non possiamo ancora rappresentarla con un singolo segmento. Infine per disegnare la curva si può procedere in due

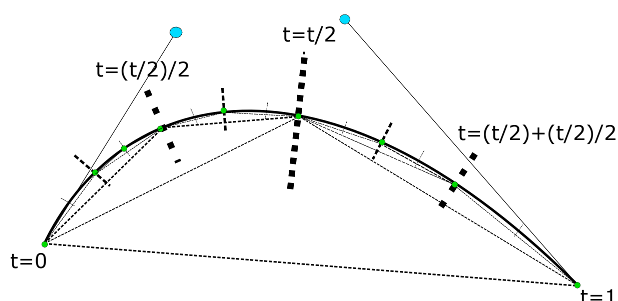

Figura 2.2: Suddivisione della curva di Bézier ricorsivamente, i punti verdi sono i punti in cui i tratti superano il test.

modi: nel primo metodo di disegno chiamiamo ripetutamente l'algoritmo di rasterizzazione di linea al verificarsi del test, oppure l'altro modo di procedere è di salvarsi la polilinea calcolata perché si può avere bisogno di effettuare operazioni e interrogazioni su di essa (come controllare quando interseca con un altra polilinea e verificare se la curva contiene auto intersezioni) prima della rasterizzazione.

2.2 Curve a tratti

Quando si vuole creare o gestire una curva per rappresentare forme complesse di solito si usano curve di Bézier di grado basso, come le cubiche o le quadratiche in successione, infatti garantendo una certa continuità tra i loro estremi si possono realizzare forme globalmente regolari e liscie.

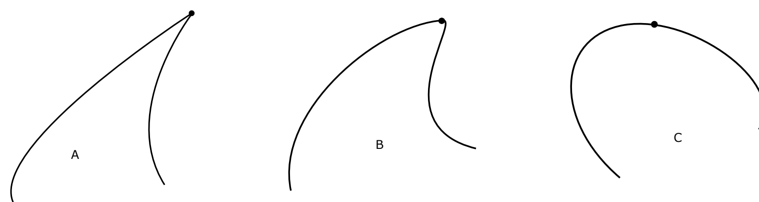

Figura 2.3: Continuità nel seguente ordine (figura A, B,C): G^0, G^1, G^2

Possiamo distinguere tre tipi di continuità:

- C^0/G^0: le due curve prese in considerazione hanno un estremo in comune.
- G^1: le due curve oltre ad avere un estremo in comune hanno anche la tangente uguale in quell'estremo.
- G^2: nell'estremo in comune hanno anche la curvatura uguale.

2.3 Offset di curve

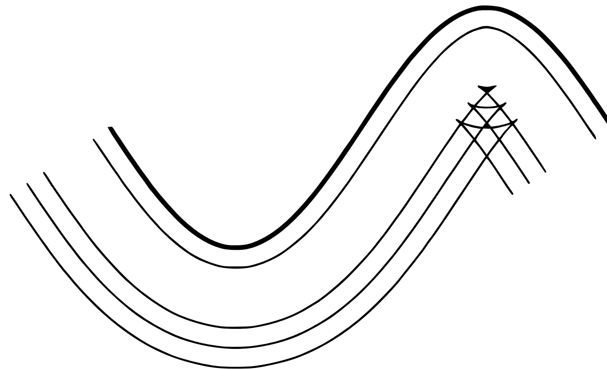

Figura 2.4: Offset della curva

L' Offset di una curva (detta curva offset) può essere definita in due modi:

1. Come l'involuzione di un fascio di cerchi congruenti centrati alla curva.
2. La curva i cui punti sono ad una distanza fissa nella direzione normale alla curva presa in considerazione.

È un aspetto molto importante che ha diverse applicazioni nell'ingegneria per lavorazione a controllo numerico in cui si deve effettuare un taglio ad una distanza fissa (utensili), o anche nella grafica 2d vettoriale che è una delle operazioni fondamentali chiamata "stroking" valutata di solito su polilinea, questo ha motivato estese ricerche che hanno concepito diverse tecniche per l'offset di una curva.

Negli anni 1990 una fondamentale ricerca svolta da Farouki and Neff stabilisce chiaramente la difficoltà nel calcolare l'esatto offset di una curva, Farouki e Sakkalis hanno suggerito che la soluzione si trova nell'odografo di una curva [6], dal quale ci ricaviamo la normale della nostra curva al momento t.

Quindi presa una curva regolare parametrica:

$$C(t) = (x(t), y(t))$$

l'odografo di una curva è dato da:

$$C'(t) = (x'(t), y'(t))$$

la tangente di una curva è invece:

$$T(t) = \frac{((x'(t)), y'(t))}{\sqrt{x'(t)^2 + y'(t)^2}}$$

Per ricavarci il suo offset che chiamiamo Cd(t) ad una distanza fissa d dalla curva regolare data:

$$Cd(t) = C(t) + d * N(t)$$

Dove N(t) è il vettore normale di C(t):

$$N(t) = \frac{(-y'(t), x'(t))}{\sqrt{x'(t)^2 + y'(t)^2}}$$

La costante d potrebbe essere negativa, in ogni caso come risultato avremo punti paralleli alla curva originale, ed inoltre anche se $C(t)$ è una curva di Bézier in genere una sua curva offset non è rappresentabile in forma di Bézier.

2.4 Disegno dell'offset di una curva

Per disegnare l'offset di una curva possiamo procedere come ho descritto nella sezione precedente "disegno di una curva di Bézier", unica differenza sta nell'utilizzare $Cd(t)$ (l'offset della curva) per ricavarci i punti da valutare,

ed il test che interrompe la ricorsione verrà effettuato quindi sul suo offset garantendo che la polilinea risultante sia la migliore possibile.

2.5 Gestione offset nei software

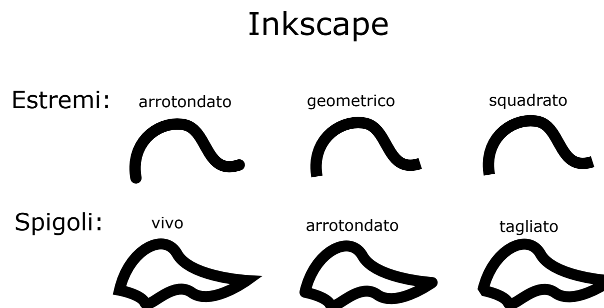

Figura 2.5: Tracciati nel software Inkscape

Prendiamo in esame "Inkscape" un famosissimo software libero per la creazione e l'editing di file vettoriali descritto nelle sezioni precedenti, la parte su cui vogliamo focalizzarci è la gestione dei tracciati di curve e linee, e come crea gli offset quando ingrandiamo quella linea o un path. Dalla guida

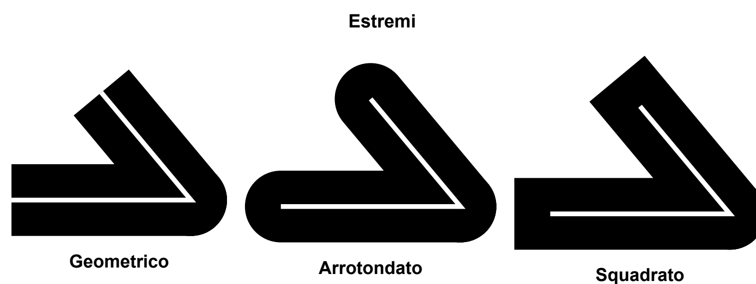

Figura 2.6: Visualizzazione nel dettaglio degli estremi

online sul software Inkscape, possiamo leggere che un tracciato è semplicemente il susseguirsi di segmenti e/o curve di Bézier, interconnessi tra di loro mantenendo le proprietà descritte prima G^0, G^1 e G^2. È facile creare tutte le forme desiderate mantenedo una certa eleganza visiva nel susseguirsi del tracciato, il programma poi permette di definire lo stile contorno con il quale

possiamo settare vari parametri, tra questi facciamo attenzione agli spigoli ed anche agli estremi che si trovano in quei tracciati continui come una curva seguita da un segmento oppure il contrario. Può qui crearsi il presupposto di volere le loro interconnessioni arrotondate o con spigolo tagliato oppure con spigolo vivo, si vede subito che un tracciato deve essere unico, cioè essere trattato nella sua interezza perché se non create con i giusti parametri le curve e le linee appariranno tagliati agli estremi e/o sovrapposte.

2.6 Pixel

Quando guardiamo una foto o una qualsiasi immagine generata su un dispositivo raster, avvicinandoci abbastanza noteremo l'immagine che prima sembrava continua iniziare a sgranarsi in tanti punti luminosi (pixel), si può quindi dire che quello che vediamo altro non è che un agglomerato uniforme di tanti piccoli segnali luminosi, la griglia è talmente fitta a tal punto da ingannarci regalandoci delle foto o immagini che sembrano perfette.

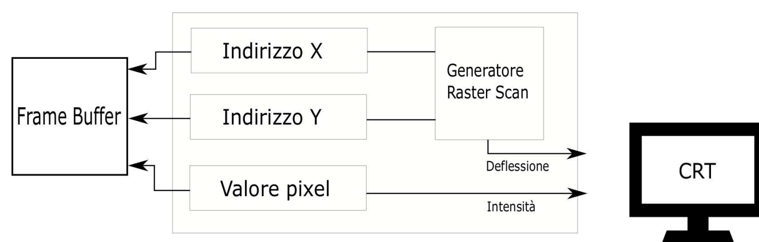

Figura 2.7: Graphics Processor Unit

Nella Grafica al calcolatore l'elemento pixel è l'unità elementare di disegno dove un'immagine su un dispositivo Raster viene generata punto per punto. Disegnare quindi consiste nel modificare l'informazione "intensità luminosa" di ogni Pixel nel display. Per accendere un pixel nella posizione (x,y) sarà sufficiente inserire nella corrispondente posizione del frame buffer l'informazione colore. Di solito attraverso lo schermo di un computer le immagini vengono ricostruite attraverso la sovrapposizione dei colori primari con "sintesi additiva", i colori primari di solito sono il rosso, il verde ed il

blu e sono chiamati RGB, per rappresentare il colore nero possiamo semplicemente assegnare il valore $rgb[0,0,0]$ invece per il bianco settiamo i valori massimi $rgb[1,1,1]$.

- In un sistema a Colori (Red, Green, Blue) avranno un valore compreso nell'intervallo [0,1] e possiamo così ottenere qualsiasi colore rappresentabile.
- In un sistema Monocromatico i valori (Red, Green, Blue) avranno lo stesso valore per tutte e tre le componenti colore.

Tutti sistemi grafici possiedono una funzione per scrivere un determinato pixel, che noi chiameremo:

$$drawPixel(x, y, r, g, b)$$

oppure

$$plotPixel(x, y, r, g, b)$$

questa funzione prende come parametri i valori x e y che sono le coordinate schermo del pixel in un sistema cartesiano a coordinate intere, ed il colore con cui vogliamo disegnare (tracciare) tale punto.

2.7 Rasterizzazione di linea vettoriale

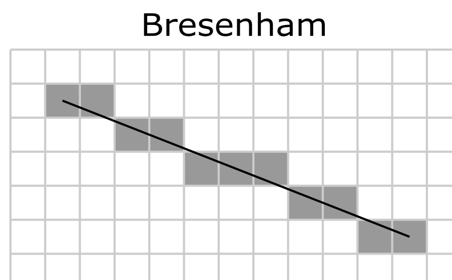

Figura 2.8: Algoritmo di linea (Jack Elton Bresenham)

Ci sono diversi algoritmi di rasterizzazione di linea, uno di questi è il DDA (Digital Differential Analyzer) che realizza operazioni con aritmetica in virgola mobile il quale comporta un arrotondamento ad un'aritmetica intera, in aiuto ci viene l'algoritmo più efficiente per rasterizzare una linea, realizzato da Jack Elton Bresenham (1962), chiamato anche "algoritmo del punto medio". Al momento è il più utilizzato anche nelle implementazioni Hardware per la sua semplicità e la bassa richiesta computazionale, è un metodo incrementale e lavora solo su aritmetica intera quindi efficente.

2.8 Riempimento di poligoni

Focalizzandoci sugli algoritmi di riempimento di poligoni, esistono svariate tecniche che permettono di rasterizzare un poligono, tra cui:

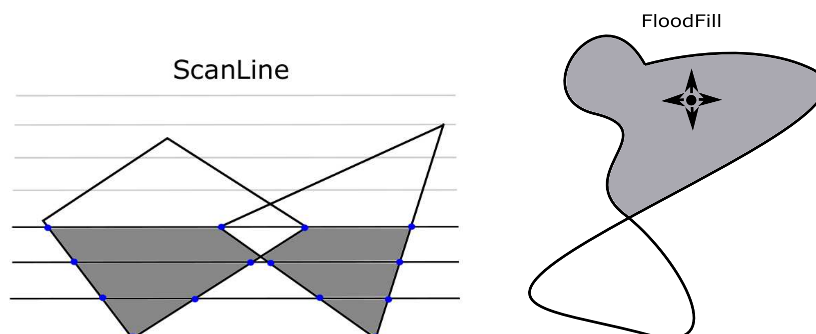

Figura 2.9: Algoritmi di riempimento di poligoni: ScanLine, FloodFill

- **Flood Fill**: è un algoritmo ricorsivo che ha bisogno di almeno tre parametri tra cui un punto interno all'area, il colore da sostituire e il colore da riempire, l'algoritmo riempie finchè non trova un confine (un colore diverso).

- **Scan Conversion**: è un altra tecnica più efficiente, ha bisogno di un poligono espresso sotto forma di segmenti e disegna le linee tracciate dalle intersezioni tra quelle parallele all'asse x e il poligono stesso, usa la regola even-odd (interno poi esterno).

- **Fill-rule**: Un misto tra le due è la fill-rule che viene utilizzata nella grafica vettoriale, consiste nel lanciare un raggio dal punto preso in considerazione verso l'infinito per qualsiasi direzione e valuta le intersezioni seguendo una di queste due regole: la regola **non-zero** inizia il conteggio da zero e ogni volta che un segmento attraversa il raggio da sinistra a destra aggiungiamo uno, invece se al contrario sottraiamo uno, alla fine se il risultato non è zero concludiamo che siamo dentro il poligono, la regola **even-odd** conta il numero di intersezioni e se questo è dispari allora il punto è all'interno se pari si trova all'esterno.

Esistono anche altre tecniche o metodi per riempire poligoni ad esempio si può utilizzare una bitmap (tessere) magari utilizzando tecniche che si basano sulla morfologia del poligono.

2.9 Bitmap

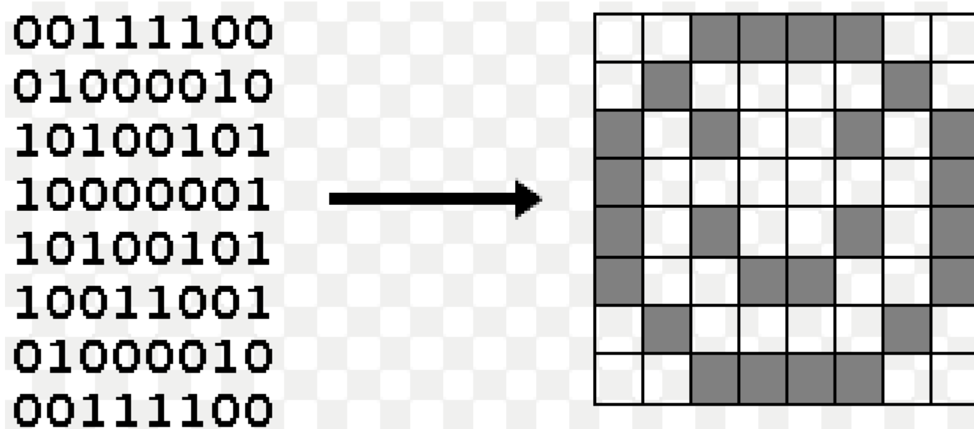

Figura 2.10: Esempio di bitmap

Le Bitmap sono immagini in bianco e nero (o se visti in una sequenza di bit avranno valore acceso o spento) tipicamente rappresentate in modalità raster.

Capitolo 3

Antialiasing

L'AntiAliasing (abbreviato "AA") è una tecnica che permette di ridurre l'effetto di aliasing (scalettatura), quando un segnale a bassa risoluzione viene mostrato ad alta risoluzione. Il termine Aliasing non è un termine usato unicamente nella computer grafica, ma anzi trova riscontro in una vasta tipologia di argomenti come l'audio, oppure la trasmissione dei segnali digitali, ma in questo capitolo ci concentreremo sulla sua controparte nel relativo ambito della grafica, durante la rasterizzazione della nostra immagine le schede video lavorano con triangoli e linee che per essere mostrati devono essere rasterizzati, è in questa fase che vengono scelti i colori dei singoli pixel; il valore sarà il colore a cui appartiene il centro del pixel ed è proprio per questo motivo che durante il disegno di linee oblique si creano zone ad alta differenza di colore (aliasing). Per capire meglio si pensi ad una scacchiera dove i vari quadratini sono bianchi e noi vogliamo disegnare sopra una linea obliqua (non parallela all'asse delle ordinate o a quella delle ascisse) però il modo di farlo è solamente riempendo totalmente un quadratino intero per volta, riempendoli fino a quando non avremo realizzato la forma desiderata. Sarà subito evidente la scalettatura e l'impossibilità di sfuggire a questo limite, quindi il compito dell'Antialiasing è quello di arrotondare le geometrie da campionare in maniera da mostrarsi più uniforme, liscia, precisa e in definitiva più apprezzabile, lo fa donando dove necessario una graduazione

di colore che inganna il nostro occhio a facendoci credere che il disegno che stiamo guardando sia perfetto. E' bene fare chiarezza che un pixel non ha un area, è stato dimostrato da Alvy Ray Smith [7] (considerato un pioniere della grafica computerizzata, ed uno dei fondatori della Pixar) che un pixel non è una piccola area quadrata, ma è un punto campione, esiste in un solo punto e per un'immagine a colori il pixel può contenere tre punti campioni, uno per ogni colore primario che contribuisce al campione.

Nel tempo sono state sviluppate svariate tecniche di AA, ognuno di esse porta con se dei vantaggi e altrettanti svantaggi, quindi la scelta di una tecnica è determinata dal bisogno e dal contesto in cui verrà applicata, le difficoltà che si incontreranno nell'implementazione di applicazioni reali sono soprattutto nell'ottenere la massima qualità possibile al minor costo, perché il loro uso più comune è proprio nei videogiochi, un ambito in cui per avere un buon impatto visivo dobbiamo tenere alti i frame per secondo (fps, almeno 30 per donare un effetto di movimento all'occhio umano) e più alti saranno, migliore sarà la fluidità dei movimenti; ogni tecnica utilizzata porta con sè un tempo di calcolo o costo in termini prestazionali, che spesso provoca il calo di questi ultimi citati.

Data la vastità di tecniche sviluppatasi nel tempo noi tratteremo le principali e ne implementeremo un paio addentrandoci in questo argomento con un occhio di riguardo verso la grafica vettoriale.

3.1 Linea di Xiaolin Wu

Quando si rasterizza una linea (come visto nel capitolo 2, algoritmo di Bresenham) bisogna passare attraverso il processo di rasterizzazione che ne determina il colore dei pixel. Per superare questo inconveniente e disegnare una linea da un aspetto fluido e liscio possiamo usare l'algoritmo di Xiaolin [8]. Prendiamo in considerazione la linea di Bresenham (figura 3.1 a sinistra), al momento della Rasterizzazione arrotondiamo semplicemente al pixel più vicino non tenendo in considerazione l'errore generato, invece nel metodo di

Figura 3.1: A sinistra si nota una linea ideale e a destra una linea dopo il processo di Rasterizzazione

Xiaolin Wu ad ogni passo si calcola la distanza tra la coordinata Y effettiva e la cella della griglia più vicina, determinando la quantità di colore diviso tra quei pixel.

Figura 3.2: A sinistra possiamo trovare Bresenham che arrotonda solamente al pixel vicino, a destra Xiaolin Wu che determina il colore tramite la distanza dal centro del pixel e la linea

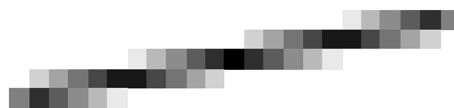

Figura 3.3: Rasterizzazione di linea con antialiasing di Xiaolin Wu's

3.2 SuperSampling

La prima tecnica di AA utilizzata, è chiamata **SSAA "Super Sampling Anti Aliasing"** detta anche FSAA "Full Scene Anti Aliasing". È sicuramente la migliore per quanto riguarda la qualità risultante dell'immagine

finale, ma è anche la più costosa per la sua richiesta computazionale (unico svantaggio).

Il SuperSampling non fa altro che renderizzare l'immagine ad una risoluzione maggiore, utilizzando poi i pixel extra per effettuare il **"DownSampling"**, che è una media dei colori che compongono l'immagine a risoluzione maggiore, riversandolo all'interno del pixel di destinazione del frame buffer alla risoluzione di output, per tutti i pixel che compongono l'immagine.

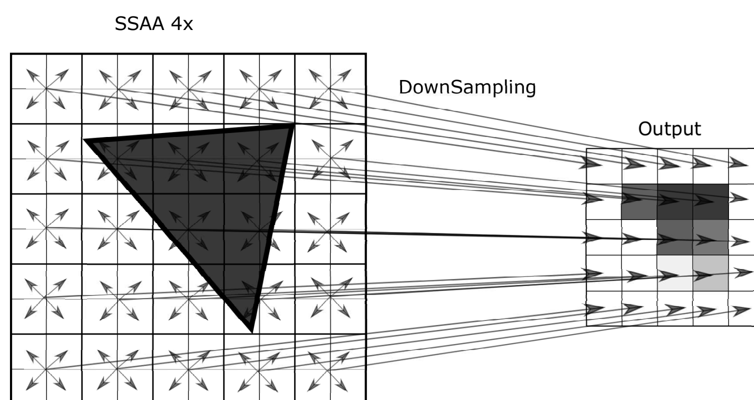

Figura 3.4: SSAA, il disegno a risoluzione maggiore viene processato nella sua interezza con il processo di "downsampling" e la media dei pixel extra sarà il colore del pixel nel dispositivo di output.

Questo approcio permette di ottenere delle immagini di ottima qualità, ma di contro porta con se un elevato costo computazionale, per questo con il tempo si sono sviluppate svariate metodologie di questa tecnica, cercando di ottenere il miglior risultato al minor costo.

3.3 MultiSampling

Quando parliamo di **MultiSampling**, abbreviato **MSAA**, ci stiamo riferendo ad un caso speciale del SuperSampling che effettua le operazioni di copertura dei campionamenti solo ai bordi delle nostre geometrie (forme), evitando così di eseguire calcoli dove non avrebbero un effetto positivo alla resa dell'immagine finale.

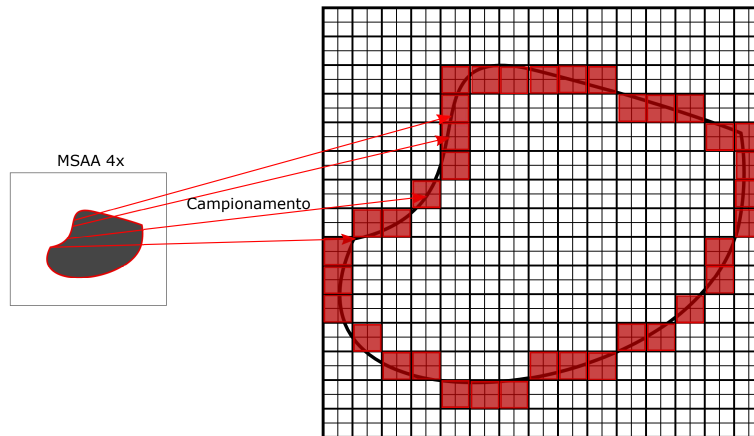

Figura 3.5: MSAA, campionamento eseguito solo sui bordi, le aree rosse.

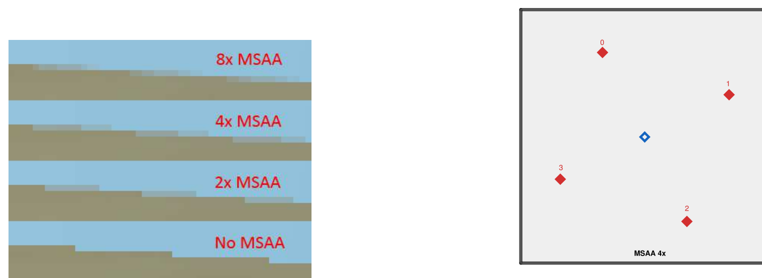

Figura 3.6: Esempi MSAA e Punti campione per MSAA 4x

Il MultiSampling è molto simile al Supersampling, perché calcola i colori risultanti del pixel di output campionandoli ad una risoluzione maggiore ma solo ai bordi delle geometrie che compongono la scena, ad esempio effettuare il supersampling nel centro di un triangolo (dove non c'è Aliasing) non porta certo una perdita di qualità ma nemmeno un miglioramento dato che il colore finale di quel pixel sarà il medesimo di quello iniziale. Una fase molto importante è il campionamento [9], i parametri relativi al posizionamento dei punti campione possono variare, infatti abbiamo varie tipologie di griglie, che variano la loro distribuzione nel subpixel.

- **Orderer Grid Super Sampling** (OGSS): in questa tecnica, la distribuzione dei punti campione è ordinata come una griglia regolare,

e risolve la maggior parte dei problemi di aliasing molto vicini ai lati verticali ed orizzontali.

- **Rotated Grid Super Sampling** (RGSS): quest'altra tecnica consiste nel ruotare la griglia dei punti campione per ovviare agli svantaggi posti dall' OGSS, anche se i pattern regolari sono ancora percepibili dall'occhio umano.

- **Random Sampling**: la distribuzione dei subpixel avviene in maniera casuale attorno al pixel, eliminando così pattern regolari.

Avendo disposto i nostri punti campione, non ci resta altro che controllare quanti di essi siano coperti dal poligono che stiamo valutando; il valore risultante ci dà la quantità di colore da disporre in quel determinato pixel; per valutare la copertura dei campione si utilizzano di solito dei triangoli [10] che sono le primitive più efficenti da valutare. Riassumendo, questa tecnica

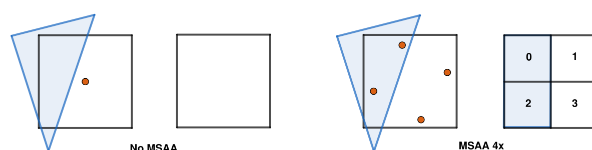

Figura 3.7: Esempio di copertura dei sotto-campioni (MSAA)

porta con sè molti vantaggi rispetto al semplice SSAA, come un calo del costo computazionale dato dal passaggio dell'elaborazione da un frame intero ai soli punti interessati dall'Aliasing, ma porta con sè altri svantaggi, perché non può agire su elementi applicati all'immagine finale, come l'illuminazione differita, texture ed altri effetti post-processing.

3.4 PostProcessing

Il MultiSampling (considerato lo standard qualitativo a cui aspirare) è stato a lungo il più utilizzato per ridurre aliasing nei giochi o nei software migliorando significamente il loro aspetto visivo, in generale quest'ultimo è più veloce del SuperSampling, ma rappresenta ancora un costo aggiuntivo significativo, ed è difficile da implementare con determinate tecniche. I metodi descritti sopra agiscono ancora prima che l'immagine sia stata completata nella sua interezza (durante il procedimento di rasterizzazione). Adesso vogliamo parlare dei metodi chiamati **PPAA** (**PostProcessing AntiAliasing**) i quali sono tutti quei filtri che prendono come punto di partenza un'immagine (una foto o un frame renderizzato, in generale un'immagine) applicando a quest'ultima tutti i calcoli di cui necessita, queste tecniche sono di facile implementazione ed in più funzionano in scenari dove MSAA non funziona (come l'illuminazione differita e altri aliasing non basati sulla geometria). La maggior parte delle tecniche Post-Processing che andremo a descrivere, sono soprattutto tecniche applicati al RealTime [11] (interessano più da vicino i videogiochi, o comunque dove abbiamo un disegno che deve essere aggiornato ripetutamente).

3.5 Morphological AntiAliasing

Il **Morphological Antialiasing** detto **MLAA** di Alexander Reshetov [12], è il precursore di una nuova generazione di tecniche di Antialiasing che rivaleggia MSAA (MultiSampling). Fa parte di quelle tecniche post-elaborazione (sull'intera immagine) che affronta alcuni svantaggi che troviamo nell'MSAA, come l'illuminazione differita e l'elevata memoria.

Per prima identifica i modelli di discontinuità, poi costruisce un percorso legando i vari bordi trovati tramite un pattern e a questo punto applica una trasparenza mescolando i colori nelle vicinanze di questi modelli per eseguire un efficace antialiasing. Quindi a differenza del precedente filtro non c'è alcun sovra-campionamento, nonostante l'idea di base sia buona ed il costo

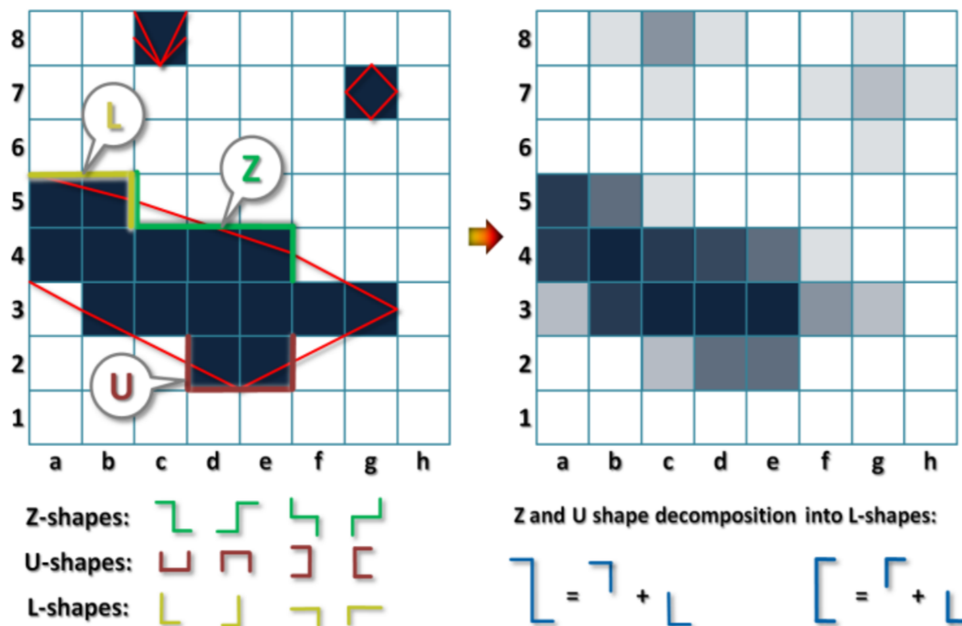

Figura 3.8: Riconoscimento forme MLAA, sulla sinistra troviamo i pattern riconosciuti (Z, L, U), sulla destra abbiamo applicato il filtro

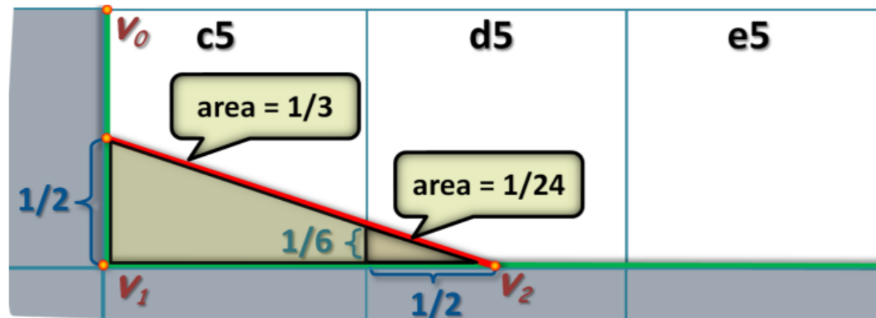

Figura 3.9: Calcolo area MLAA, come si può notare la seguente forma viene divisa in mezzo e viene gestita come area di trapezoidi

accettabile, non viene utilizzato molto in quanto non riesce a gestire bene i bordi spessi ed è causa di una possibile distorsione alle texture.

3.6 Conservative Morphological AntiAliasing

In soccorso alle problematiche poste dall'MLAA parleremo ed approfondiremo una nuova tecnica chiamata **CMAA** (**Conservative Morphological AntiAliasing**) che può essere usata in una vasta gamma di applicazioni, compresi scenari come testo, pattern ripetuti, determinate geometrie (linee elettriche, recinzioni di mesh, fogliame) ed immagini in movimento. Il Conservative Morphological Antialiasing, rispetto ad altre tecniche ppaa offre una stabilità temporale migliore, e si basa su un algoritmo che gestisce solo discontinuità simmetriche (più prudente) per evitare sfocature indesiderate. Può funzionare in modo efficente anche su schede grafiche di fascia medio-bassa e può essere accettabile come sostituto dell' "MSAA 2X".

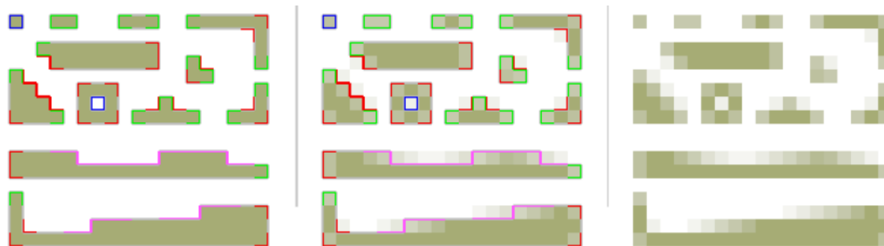

Figura 3.10: CMAA, esempi delle forme riconosciute.

Il CMAA possiede 4 passaggi logici di base, non necessariamente corrispondente all'ordine di implementazione:

3.6.1 Analisi dell'immagine per discontinuità di colore

I bordi vengono rilevati confrontando i pixel adiacenti, dove troviamo un fronte di discontinuità, cioè il valore della distanza del colore è superiore ad un limite prefissato, ciò vuol dire che ha trovato un bordo oppure un artefatto. La soglia di discontinuità viene scelta empiricamente (possiamo usare il calcolo della distanza Euclidea) ed i bordi trovati vengono memorizzati in un buffer locale. Questo passaggio non è "esclusivo" del CMAA, ma è condiviso da molte altre tecniche post-processing.

3.6.2 Estrazione dei bordi localmente dominanti

Utilizzando un piccolo kernel, questa parte si occupa di fare una potatura dei bordi rilevati, per ciascun bordo rilevato nel passaggio precedente, viene confrontato con i 12 bordi più vicini, escludendo così dai bordi quelli che hanno un colore che superano una certa soglia, riducendo quindi l'interferenza con bordi meno evidenti.

3.6.3 Gestione di forme semplici

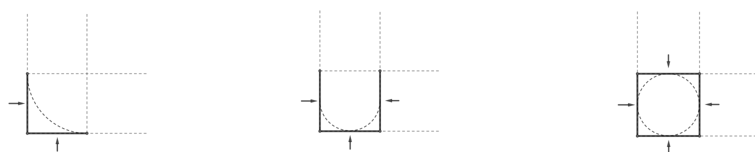

Figura 3.11: Gestione delle forme a 2, 3, 4 bordi

I bordi rilevati nel primo passaggio e perfezionati nel secondo vengono utilizzate per formulare delle ipotesi, che servono a verificare l'esistenza di forme con 2, 3, 4 bordi, quando vengono rilevati bisogna versare una parte del colore esterno verso l'interno dai pixel adiacenti, la quantità di colore è calcolata empiricamente.

3.6.4 Gestione della forma Z

Successivamente cercheremo la forma Z per tutti e quattro gli orientamenti con differenza di 90 gradi, questa forma si può allungare in maniera simmetrica da entrambe le estremità se esistono i bordi necessari. La nostra "Z" trovata nella sua angolazione ci darà una linea verticale o orizzontale in cui andremo a riversare una percentuale di colore che viene dal pixel adiacente al bordo (esterno).

Per il calcolo della quantità di colore da miscelare possiamo verificarne l'area formata dalla forma Z (una soluzione potrebbe essere il calcolo dell'area che forma la diagonale che connette i bordi esterni).

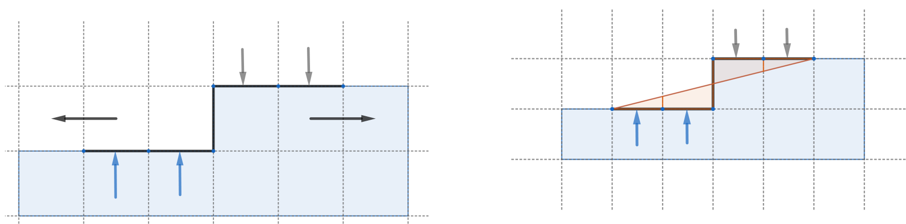

Figura 3.12: Forma Z nel CMAA, sulla sinistra Gestione della forma Z espandibile, sulla destra la determinazione dell'area occupata dalla forma Z

Capitolo 4

Sperimentazione Numerica

4.1 Ambiente di Calcolo

Durante lo studio delle tecniche viste nei precedenti capitoli si è sentito il bisogno di implementare una base su cui effettuare le nostre prove ed implementazioni (il nostro studio si baserà principalmente nell'analisi di pixel o sub-pixel e lo studio che porta alla visualizzazione di essi con o senza filtri), si è cercato un sistema che non faccia già da sè calcoli nascosti a migliorarne la qualità finale. La scelta dell'ambiente di lavoro è caduta su "Eclipse Standard Edition" (il programma è scritto in Java quindi esiste per molte piattaforme come Linux, HP-UX, AIX, macOS e Windows, ed è software libero distribuito sotto i termini della "Eclipse Public License") e come linguaggio di programmazione abbiamo usato Java, invece come ambiente grafico abbiamo scelto di utilizzare una libreria chiamata jogamp (cross platform java) che provvede a fare il bindings alle OpenGL.

Come primo passo si è deciso di impostare il progetto in modo da agevolare la creazione della fase di Sperimentazione, usando le possibilità di astrazione che mette a disposizione il linguaggio Java, ci siamo creati una classe astratta che reinstrada le chiamate principali che mette a disposizione la libreria per la fase di rendering (come dispose(), init(), display(), reshape(), gli eventi del mouse,...) così da crearci un sistema che viene costruito con

pochissimi passi e che abbia le funzionalità utili al nostro scopo (facilmente modificabili), per utilizzare le potenzialità descritte ci basta estendere la classe astratta (chiamata "ViewerAbstract") ed implementare i loro metodi astratti con delle semplici istruzioni.

Nello specifico tale classe astratta viene estesa se si vuole creare una nuova finestra ed ogni View creata può anch'essa creare altre finestre a sua volta, l'accesso alla memoria delle texture è realizzato tramite un Singleton (design pattern) con un sistema che fa da gestore tra le immagini in memoria già caricate e la richiesta di utilizzo di una nuova immagine, così da non dover caricare le stesse texture una per ogni finestra. È possibile scegliere la risoluzione iniziale della finestra e se essa sarà fissa o ridimensionabile, possiamo anche scegliere se il nostro programma eseguirà il rendering a scadenza fissa (intervalli di tempo 30 o 60 fps) oppure se settiamo gli fps a 0 viene utilizzata la modalità in cui esegue un rendering solo al verificarsi di un evento (ad esempio mouse, tastiera, ridimensionamento della finestra, GUI), mettiamo a disposizione oggetti per la realizzazione degli elementi dell'inferfaccia grafica che permettono di modificare i valori durante l'esecuzione di un Test.

Sulla base di questa implementazione si vuole dare alla libreria la possibilità di accedere al pixel, leggere o scrivere un colore in una determinata posizione, per far questo data la difficoltà incontrata ad accedere singolarmente al pixel (OpenGL), si è creato una Classe che simula un frame buffer mettendo a disposizione funzioni come draw_pixel(x,y,color), get_pixel(x,y), clearColor(color). Infine quando abbiamo finito di applicare le nostre modifiche si può chiamare la funzione draw_pixels(GL2 gl) che disegna nella sua totalità l'immagine appena realizzata sul frame buffer reale.

4.2 Implementazione degli algoritmi base

Nel capitolo 2 parliamo dell'insieme degli algoritmi che fanno parte della computer grafica, il nostro compito è stato di implementarli per creare un

ambiente per il rendering di curve, path di curve e il relativo spessore (Stroke) e riempimento (Fill) .

4.2.1 Bitmap

Una "Bitmap" è una sequenza (array di bit) di valori indicanti se è acceso o spento, la loro metodologia è basata sul "Raster", a volte il termine bitmap infatti viene usato per indicare proprio una grafica Raster. L'oggetto che abbiamo implementato nella libreria può essere immaginato come un ritaglio posizionato sul frame buffer (almeno a livello logico), abbiamo realizzato un insieme di funzioni (in parallelo agli algoritmi che scrivono sul frame buffer) che permettono di disegnare (calcolando in maniera automatica il ritaglio necessario a contenere la curva o il disegno) linee su bitmap e riempire forme, permette di effettuare operazioni booleane tra bitmap e infine possiamo anche scrivere sul frame buffer la bitmap definita. È uno strumento che può essere molto utile in certe operazioni, difatti come spiegherò più avanti verrà utilizzata come base per un possibile algoritmo di riempimento.

4.2.2 Rasterizzazione di linea Vettoriale

Appoggiandoci al nostro frame buffer che come spiegato in precedenza, permette di accendere e spegnere o settare un colore ad un pixel, abbiamo implementato un paio di algoritmi di rasterizzazione di linea come il DDA, algoritmo di Bresenham, riscontrando un effettivo aumento delle prestazioni nell'utilizzo del secondo, il loro lavoro consiste nell'accendere una sequenza continua di pixel che va da $P1$ a $P2$, l'effettivo costo in più nel DDA è nell'utilizzo di calcoli in virgola mobile, invece che in aritmetica intera come utilizzato nella linea di Bresenham.

Come è possibile evincere dalle seguenti figure (parte sinistra, Bresenham o DDA) presi dal programma da noi realizzato, rasterizzando una linea obliqua si può vedere la scalettatura, perché obbligati ad arrotondare al pixel più vicino. Per ovviare a questo problema abbiamo realizzato la linea (con

Figura 4.1: Rasterizzazione di linee, a sinistra Bresenham, nel centro Xiaolin Wu e a destra Gupta Sproull

Antialiasing) di Xiaolin Wu come mostrato nella figura (parte centrale), si evince una linea più liscia e fluida ed inoltre esiste un altro metodo di rasterizzazione l'algoritmo di Gupta Sproull che però come si vede nella parte destra rende la linea più spessa.

4.2.3 Rasterizzazione di Curve di Bezier

Nella grafica al computer è una delle primitive più utilizzate, per disegnare una curva di Bézier ci si può appoggiare alla rasterizzazione di linea chiamandola ripetutamente sui punti ricavati dall'algoritmo. Per poter realizzare forme complesse si usano più curve posizionate in sequenza, abbiamo così realizzato un path tramite semplici curve cubiche, nel programma di test utilizziamo una sintassi per definire la continuità in G^0, G^1, G^2. Utilizzando la nostra libreria abbiamo creato un editor che permette la realizzazione di più path e di estrarne i control point copiandolo semplicemente negli appunti di sistema. L'editor successivamente ha avuto una funzione base per la realizzazione dei disegni vettoriali futuri, tra le funzioni utili sono state create quelle che estrapolano una sequenza di curve di Bézier cubiche dalla specifica sintassi realizzata per esprimere una sequenza con continuità diverse. Come si può notare dalle figure mostrate, l'interfaccia permette di creare delle sequenze di curve (path) o più path contemporaneamente (per poterli posizionare nell'insieme), selezionarle o modificarle, si possono poi

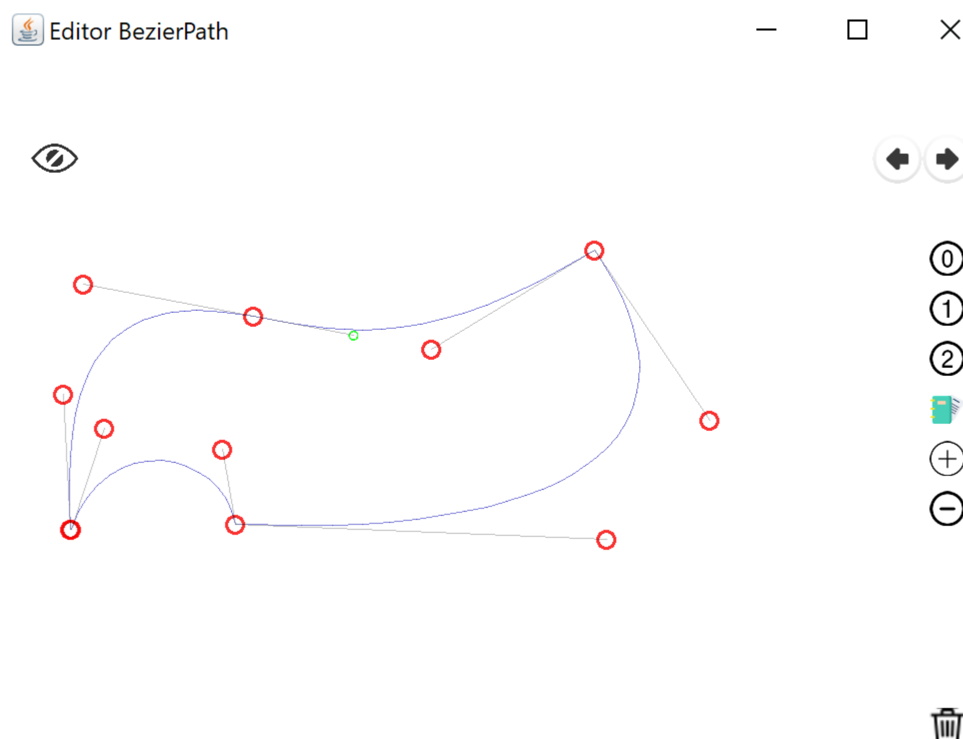

Figura 4.2: Editor Bezier Path

nascondere i control point se vogliamo focalizzarci sulla curva stessa. Per la realizzazione dell'interfaccia grafica si è pensato di aggiungere alla libreria tre tipi di tasti interattivi, ogni tipo di tasto ha delle proprietà in comune, un tipo di tasto può avere un'immagine (con trasparenza o meno) nello stato "rilasciato" ed una seconda immagine se "premuto", possiamo effettuare su di essi tutte le trasformazioni di scala, rotazione e traslazione. Quando creiamo uno di questi oggetti si ha la possibilità di specificare (Override) il loro coportamento al sollevarsi di un evento su di essi, tali metodi vengono sollevati in modo automatico dal sistema implementato.

- **Button**: è un tipo di tasto che memorizza la scelta in modo statico, cioè salva il suo attuale stato (può essere visto come acceso o spento) e si possono specificare le azioni da compiere negli Override degli eventi OnPressed e OnReleased.

- **Key**: simile al button nella sua specifica (definizione) ma con un ciclo diverso, infatti possiamo immaginarlo come un tasto analogico (ad esempio il tasto di una tastiera) che quando premuto solleva l'unico evento OnClicked (ad ogni step o frame rilancia l'evento), invece quando viene rilasciato non solleva nessun evento.

- **Knob**: questo oggetto ha un comportamento lievemente diverso dagli altri, nello specifico è un oggetto trascinabile e ad ogni modifica solleva un evento chiamato OnTry che ci da l'informazione della nuova posizione dell'oggetto.

4.2.4 Riempimento poligoni

Per la libreria messa appunto, il passo fondamentale adesso sarà di dare la possibilità di riempire le forme espresse tramite linee e curve, come accennato nel capitolo 2 esistono infatti vari modi per fare questo: possiamo usare un algoritmo ricorsivo come la Flood Fill, ma è alquanto inefficiente, come risultato è molto limitato in quanto non può riempire forme come il numero otto in una sola chiamata, avevamo accennato del Fill-rule e infine la "Scan Conversion" che sarà quello utilizzato nella maggior parte dei test effettuati. Ho esplorato l'implementazione anche di un altro metodo basato sulla bitmap, questo metodo segue il riempimento in base alla forma impressa nella sequenza di bit, ma visto che è meno efficiente (in quanto ha bisogno di passare la bitmap con il disegno da riempire) si è poi deciso di utilizzare la Scan Conversion come metodo di default.

4.3 MultiSampling (MSAA)

Il MultiSampling, si basa sulla stessa filosofia del SSAA, ma rispetto a quest'ultimo risulta molto meno costoso, perché effettua i calcoli solo ai bordi delle geometrie. Prima considerazione da fare per la fase implementativa del MSAA è l'ordine di disegno, perché esso potrebbe influire negativamente

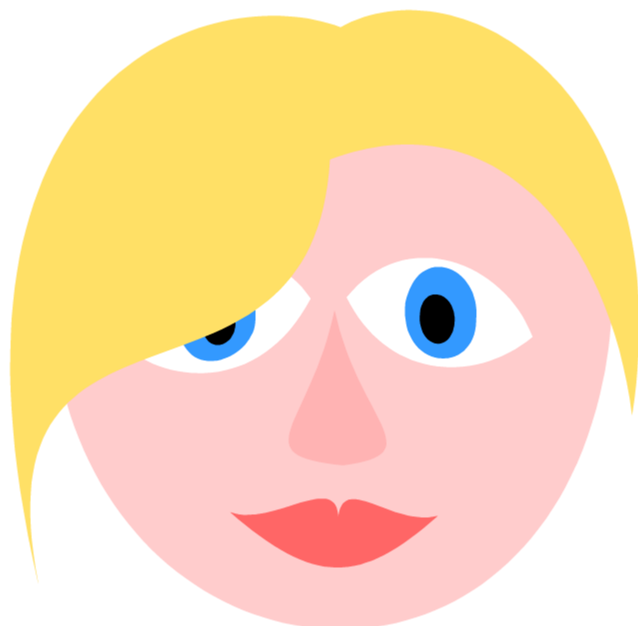

Figura 4.3: MSAA 16x

sulla resa finale, ad esempio se il bordo di un poligono è coperto da un altro, quando andiamo ad applicare il filtro sul disegno che sta dietro, dobbiamo ricordarci di tagliare il poligono per non applicare l'effetto dove non dovuto, oppure applicare il filtro subito dopo il disegno così da sovrascrivere le parti non visibili al nostro disegno successivo. Dato che applichiamo la stessa filosofia del SSAA ma solo ai bordi, avremo bisogno del poligono calcolato a risoluzione maggiore ma anche quello a dimensioni normali, una variabile (intera per semplicità) che indica la scalatura dalla geometria originale ed il colore del poligono da renderizzare.

Possiamo suddividere in più fasi l'applicazione di questa tecnica:

4.3.1 Valori di input

In questa prima fase dato in input il poligono a risoluzione corrente ci ricaviamo quello a risoluzione maggiore. Abbiamo usato le curve di Bézier cubiche, che dati i control point generano la curva con approssimazione al

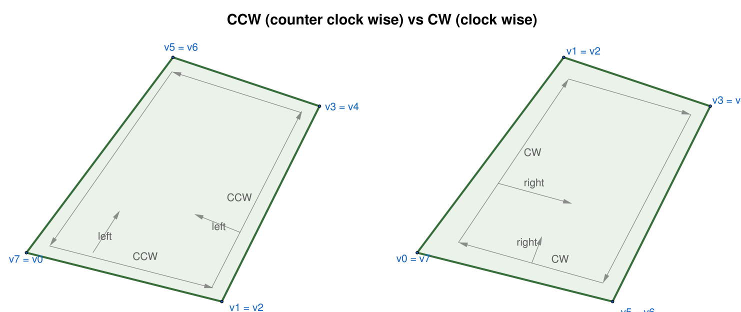

Figura 4.4: Sulla sinistra troviamo una definizione di linee nel verso antiorario, e sulla destra nel verso orario

pixel, le linee generate dal nostro algoritmo di Bézier devono essere ordinate, una proprietà che ci viene comoda nelle fasi successive. Inoltre è possibile applicare alla curva trasformazioni geometriche come traslazione, rotazione, scala ed è grazie a queste proprietà di cui gode che riusciamo facilmente a generare la nostra curva alla risoluzione maggiore, applicando la trasformazione di scala ai control point e chiamando da questo punto la creazione della curva.

4.3.2 Supporto al campionamento

Come seconda fase, creiamo il supporto per la fase di campionamento, quello di cui abbiamo bisogno è un meccanismo che permetta di valutare se il punto campione che stiamo attuamente considerando sia dentro il poligono (a risoluzione maggiore) o sia all'esterno, così da permettere di determinare la media del colore finale a risoluzione normale. Per il supporto alla fase di campionamento possiamo optare per una triangolazione del poligono, abbiamo scelto una via di mezzo dove triangoliamo solo i bordi del poligono e lo facciamo mantenedone l'ordine così da riuscire ad indicizzare approssimativamente la posizione di dove si trova il triangolo per valutare la copertura

dei campioni.

4.3.3 Triangolazione bordi

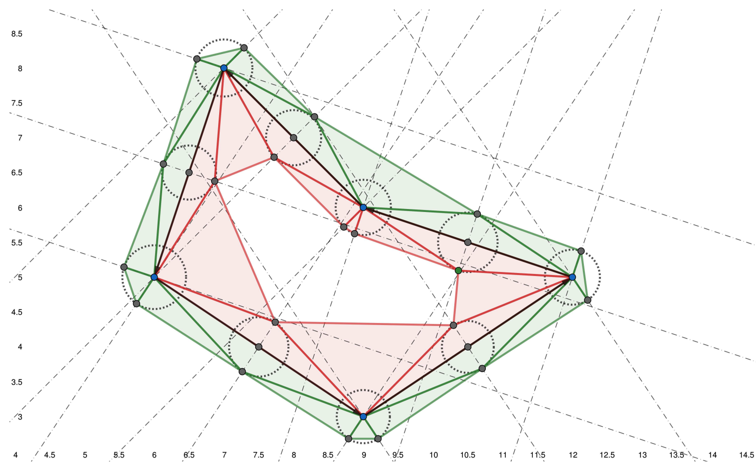

Figura 4.5: Esempio di triangolazione dei bordi interni ed esterni

In questa fase vogliamo, dato un poligono in formato di linee (Lines) ordinate, ricavarci una serie di triangoli che approssimano la parte piena del bordo. Il focalizzarci sul bordo permette prima di tutto di ridurre la complessità a quello che sarà la vicinanza di essa, data la necessità di avere le linee ordinate ne sfruttiamo la proprietà per ridurre le soluzioni a due tipi di inserimenti bordi , Destra o Sinistra, quello che stiamo andando a creare è un triangolatore di bordi per il campionamento, i triangoli di cui abbiamo bisogno si trovano o al di fuori del poligono o all'interno; stabilendo adesso che un poligono definito in "Counter Clock Wise" (CCW) sarà pieno alla sua sinistra durante tutto il suo percorso, al contrario se è espresso in senso orario "Clock Wise" (CW) il triangolo sarà a destra.

Per l'implementazione si è deciso di valutare ogni linea guardando alla linea successiva, di valutare se siamo in CCW o CW e se quindi stiamo riempendo a destra o sinistra. data la specularità dell'implementazione se definiamo un poligono CCW riempiamo a sinistra se combacia con quello settato da noi, altrimenti crea i triangoli del lato opposto. Come prima funzione

utile alla risposta se la prossima curva è a destra o a sinistra, ci avvaliamo nel valutare l'angolo di direzione delle due linee prese in considerazione: A1 è l'angolo della linea 1, A2 è l'angolo di direzione della linea 2, il test si occupa principalmente di misurare le distanze.

$$boolean\, LEFTorRIGHT(angolo1(A1), anglolo2(A2))$$

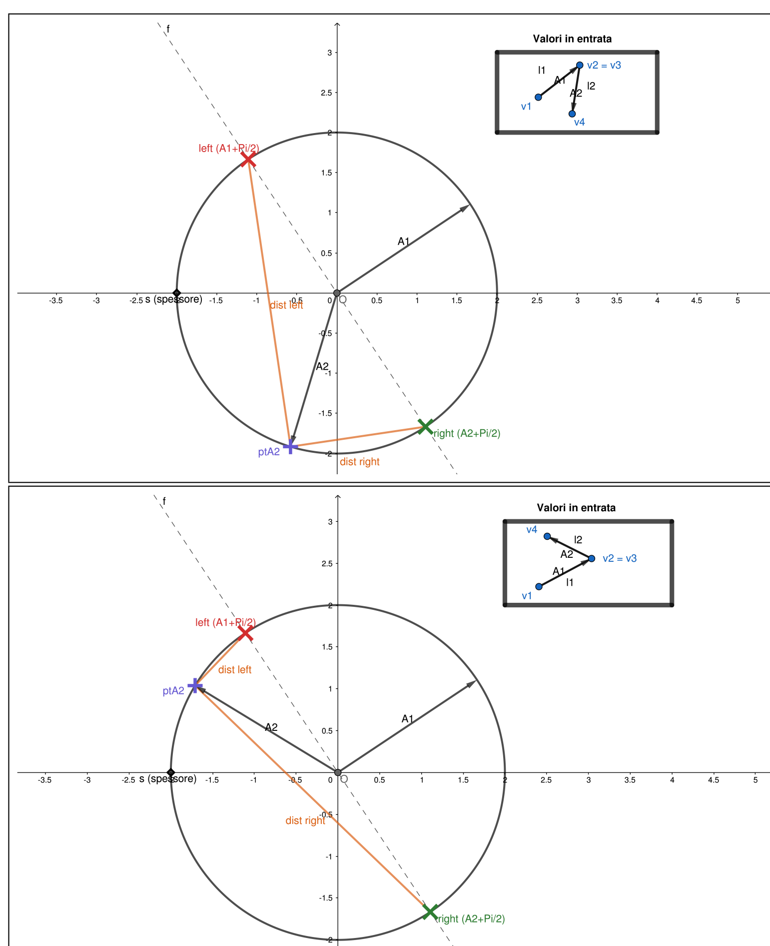

Figura 4.6: Rappresentazione della funzione LEFTorRIGHT

La funzione LEFTorRIGHT prende in input l'angolo a cui punta la linea 1 e la 2 (A1, A2) e ritorna un valore booleano che indica se ci troviamo di fronte ad una linea verso sinistra o una linea verso destra.

Generiamo come primo elemento il punto ad una distanza c con angolo A2 dall'origine (della seconda linea),

$$pA2x = c * cos(A2)$$

$$pA2y = c * sin(A2)$$

quì vogliamo trovare i punti intersecanti la circonferenza con la retta perpendicolare della linea 1 (retta passante per l'origine), a questo punto possiamo proseguire creando l'angolo che rappresenta la retta, per fare questo possiamo aggiungere $\pi/2$ (90°) per ricavare l'angolo verso sinistra , e -$\pi/2$ (90°) verso destra.

$$nA = \pi/2$$

$$pLeftx = c * cos(A2 + nA) \qquad pLefty = c * sin(A2 + nA)$$

$$pRightx = c * cos(A2 - nA) \qquad pRighty = c * sin(A2 - nA)$$

i due punti trovati dovranno essere valutati con pA2, chi avrà distanza minore deciderà il risultato determinando se siamo di fronte ad una svolta a sinistra o a destra.

$$dR = distance(pRightx, pRighty, pA2x, pA2y)$$

$$dL = distance(pLeftx, pLefty, pA2x, pA2y)$$

Se $dL < dR$ allora siamo in presenza di una direzione verso sinistra altrimenti se $dR < dL$ allora siamo in presenza di una direzione verso destra.

La distanza tra due punti la calcoliamo semplicemente chiamando la funzione $float\, distance(pt1, pt2)$come segue:

$$x = |p1X - p2X| \qquad y = |p1Y - p2Y|$$

$$d = \sqrt{x^2 + y^2}$$

Per l'angolo tra i due punti invece la funzione è $float\, angle(pt1, pt2)$

$$a = \arctan \frac{pt2y - pt1y}{pt2x - pt1x}$$

se il risultato è $pt2x - pt1x < 0$ al risultato aggiungiamo (180°) $a = a + \pi$.

La nostra funzione ha uno sviluppo speculare, che cambia in base a dove vogliamo riempire se è a sinistra ($+nA = Pi/2$) o a destra ($-nA = Pi/2$), possiamo quindi dividere i cicli in maniera simmetrica, se siamo in una curva verso sinistra generiamo i triangoli a sinistra in una curva definita in CCW, a destra se CW, se siamo in una curva verso destra riempiamo a sinistra in CCW altrimenti a destra, come possiamo notare dal codice di esempio i vari casi saranno gestiti nei seguenti rami (*1,*2,*3,*4 sono i vari casi gestiti dal triangolatore).

Listing 4.1: Pseudo codice, "Triangolatore bordi"

```
//siamo in direzione verso sinistra
if(LEFTorRIGHT(A1,A2)==Left) {
        if(CCW) {Triangoliamo a sinistra}        //*1
        else if(!CCW) {Triangoliamo a destra}    //*2
        }
//siamo in direzione verso destra
else if (LEFTorRIGHT(A1,A2)==Right){
        if(CCW) {Triangoliamo a sinistra}        //*3
        else if(!CCW) {Triangoliamo a destra}    //*4
        }
```

4.3.4 Riempimento a sinistra (*1)

Con $v1, v2, v3, v4$ le linee che fanno parte di quel pezzo $l1(v1, v2)$ e $l2(v3, v4)$ valutiamo il punto medio $m1 = (v1 + v2) * 0.5$, e $m2 = (v3 + v4) * 0.5$.

Creiamo la funzione che ritorna un punto del triangolo con spessore verso l'angolo di una normale positiva (sinistra) o negativa (destra) dal punto $m1$ e $m2$ con la $nA1 = A1 + \pi/2$ perché triangoliamo a sinistra:

$$point\, gen(point, spessore, a)$$

Non fa altro che ritornare il punto (x, y) traslato di uno spessore verso l'angolo (a).

$resultX = pointX + (spessore * cos(a)) \qquad resultY = pointY + (spessore * sin(a))$

Ci troviamo ad applicare uno spessore verso sinistra quindi chiameremo la funzione $gen()$ nella normale positiva ,

$$g1 = gen(m1, spessore, A1 + nA)$$

stessa cosa per la linea 2:

$$g2 = gen(m2, spessore, A2 + nA)$$

qui si creano due casi uno quando siamo in presenza di un angolo estremamente stretto (acuto) o un caso normale:

1. Riempimento a sinistra in curva a sinistra (angolo acuto)

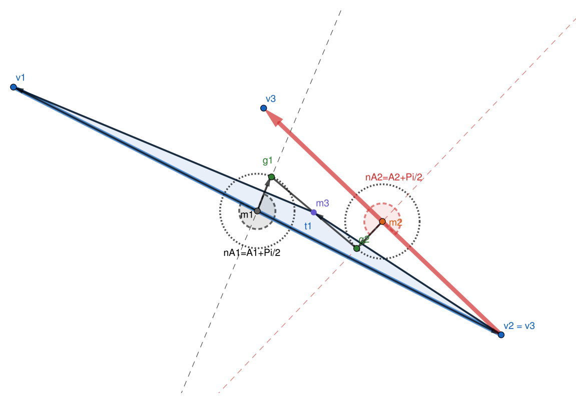

Figura 4.7: Triangolazione dei bordi verso sinistra con riempimento a sinistra (angolo acuto)

Con un angolo acuto abbiamo meno triangoli ma dobbiamo triangolare con il punto medio $(m3)$ tra $g1$ e $g2$ con particolare attenzione che alla prossima iterazione ricordiamo questo punto e non ne generiamo uno nuovo $t1(v1, v2, m3)$.

2. Riempimento a sinistra in curva a sinistra (angolo normale)

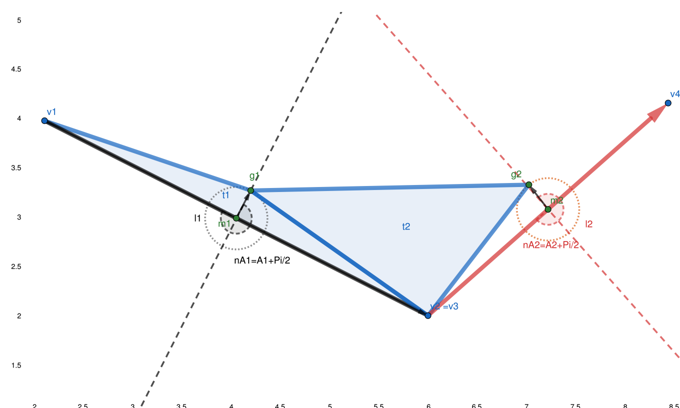

Figura 4.8: Triangolazione dei bordi verso sinistra con riempimento a sinistra (angolo normale)

Con angolo non stretto creiamo due triangoli nella seguente aggregazione $t1(v1, v2, g1), t2(v2, g2, g1)$.

4.3.5 Riempimento a destra (*2)

con $v1, v2, v3, v4$ le linee che fanno parte del pezzo $l1(v1, v2)$ e $l2(v3, v4)$ valutiamo il punto medio $m1 = (v1 + v2) * 0.5$, e $m2 = (v3 + v4) * 0.5$ qui generiamo molti più punti di spessore, 2 per ogni angolo, creiamo come sempre $g1, g2$ però con la $-nA$, ma ci calcoliamo anche $g3$ e $g4$ con angolo $(A1 - nA)$ e $(A2 - nA)$ dai rispettivi punti $v2$ e $v3$.

Anche qui si creano due casi uno con angolo stretto (acuto) e il caso normale:

1. Riempimento a destra in curva a sinistra (angolo stretto)

Nel caso con angolo acuto calcoliamo g5 e g6 chiamando la funzione $gen(conv2, s, A1, ev3, s, A2)$ ci troveremo il punto medio $(m4)$ e sarà così triangolato $t1(v1, v2, g1), t2(g1, g3, v2), t3(g3, m3, v2), t4(m3, g4, v3), t5(g4, g2, v3)$.

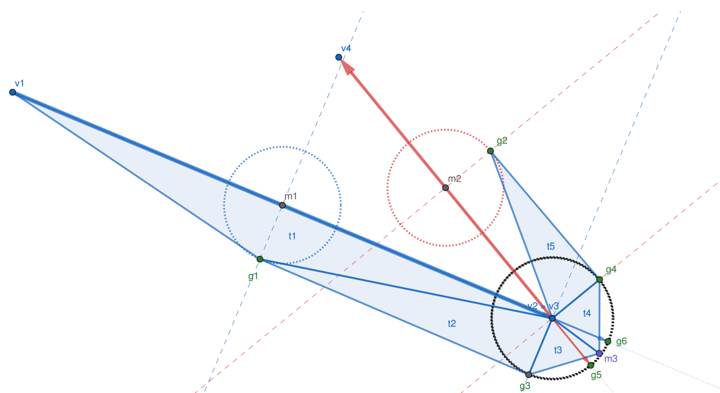

Figura 4.9: Triangolazione dei bordi verso sinistra con riempimento a destra (angolo acuto)

2. Riempimento a destra in curva a sinistra (angolo normale)

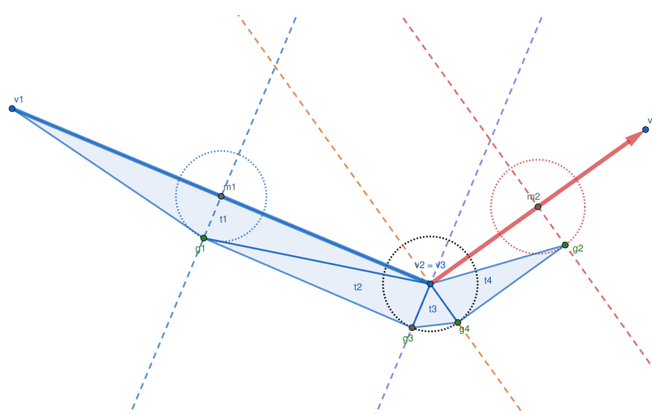

Figura 4.10: Triangolazione dei bordi verso sinistra con riempimento a destra (angolo normale)

Nel caso in cui siamo in presenza di un angolo normale aggiungeremo i seguenti triangoli $t1(v1, v2, g1), t2(g1, g3, v2), t3(g3, g4, v3), t4(g4, g2, v3)$.

I casi sono in poche parole applicabili in maniera opposta nella direzione verso destra($*3$ come $*2$, $*4$ come $*1$).

Ricordiamoci sempre di iniziare un pò prima dell'inizio (iniziare dagli ultimo elemento) e di fermarsi un pò dopo(finire sui primi elementi), con un

entrata a vuoto per configurare se siamo in presenza di angoli acuti o meno, successivamente il ciclo di connessioni ognuno guardando se stesso con il prossimo lato (se stesso con $lato(i)$ con il prossimo $lato(i+1)$), creando una striscia continua di triangoli che approssima i bordi interni o esterni da poter controllare quando siamo nella fase di campionamento.

4.3.6 Campionamento

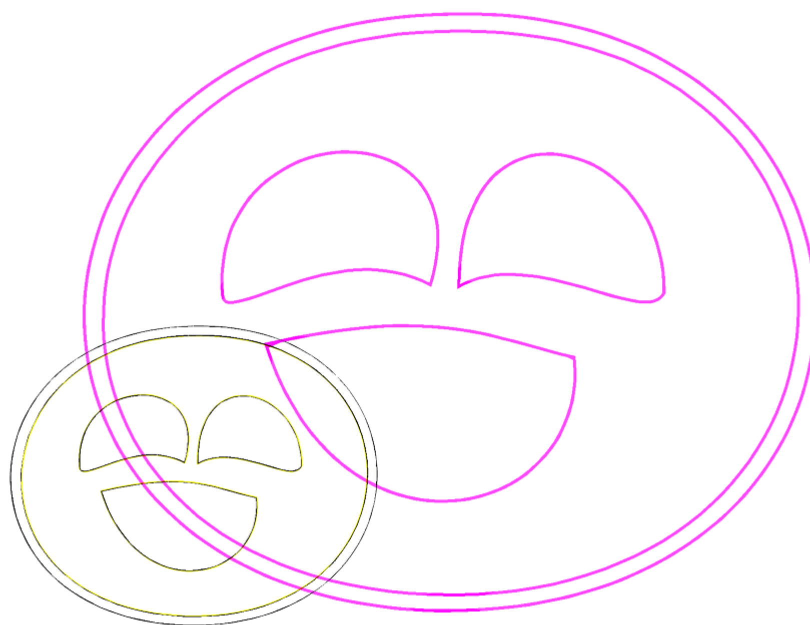

Figura 4.11: Nella figura possiamo vedere l'applicazione dell'MSAA solo ai bordi, verificando la copertura dei punti campioni sulla serie di triangoli creati a risoluzione maggiore (di colore viola)

Nella fase di campionamento come abbiamo accennato in precedenza circumnavigheremo tutto il bordo del poligono a risoluzione normale eseguendo i sotto-campionamenti con quello a risoluzione maggiore (come in figura 4.11), per questa fase ho scelto di utilizzare l'algoritmo di Bresenham modificando alcune sue parti, quando si rasterizza una linea verticale effettuiamo il

campionamento non solo al pixel in questione ma anche agli adiacenti

$$(x, y), (x + 1, y), (x - 1, y)$$

lo stesso vale per la rasterizzazione in orizzontale dove applichiamo i calcoli anche agli adiacenti

$$(x, y), (x, y + 1), (x, x - 1)$$

perché non sappiamo a priori su quale dei tre pixel avverrà l'effettiva modifica.

La nostra implementazione prevede una scalatura per un valore x, valutando un numero di campioni (xx) che calcoliamo nel seguente modo, $xx = x^2$ dove $x > 1$, quindi per $x = 2$ il nostro numero di campioni è $xx = 4$ (4 sotto-campionamenti), per $x = 3$ allora $xx = 9$ (9 sotto-campionamenti), e via via procedendo. Per il posizionamento dei campioni, si dispone ogni sotto-

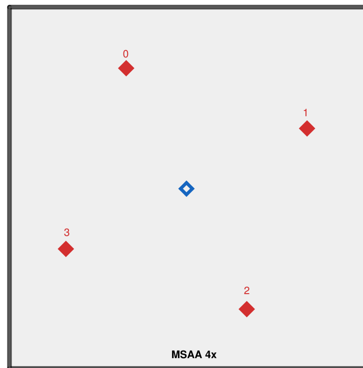

Figura 4.12: Punti campione per MSAA 4x

campione ruotato leggermente dal centro (media delle posizioni dei sotto-campioni), in seguito non facciamo altro che contare quanti sotto-campioni (sc) si trovano all'interno del poligono su un totale di sotto-campioni (tc).

Nell'esempio il poligono p1 copre due sotto-campioni su un totale di 4, allora come valore di campionamento avremo

$$campionamento = \frac{sc}{tc} = \frac{2}{4} = \frac{1}{2}$$

che sarà settato sul canale Alpha del colore della geometria.

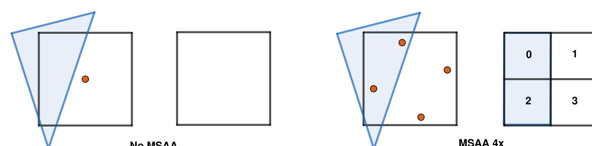

Figura 4.13: Esempio di copertura dei sotto-campioni (MSAA)

4.3.7 Bilanciamento colori

Infine possiamo applicare il valore calcolato dal campionamento nel colore della geometria, che verrà usato per disegnare sul framebuffer, modifichiamo il canale alpha con il valore risultante dal campionamento così da applicare una trasparenza con il colore che si trova al di sotto. Per facilitare la comprensione possiamo immaginare di avere un colore di sfondo (back) e uno della geometria (front), a questo punto possiamo decidere se applicare lo sfondo (back) alla geometria (front) o il contrario, nel primo caso il valore che andremo ad inserire nel canale alpha del componente back è $1 - Campionamento$, e nel secondo è semplicemente il $Campionamento$ (Campionamento è un valore compreso da 0 a 1, ed indica la quantità della componente colore che comparirà in quel determinato pixel).

4.3.8 Considerazioni finali

Il multisampling è un evoluzione del supersampling, punta a calcolare solo le aree dove è possibile trovare aliasing quindi ai bordi delle geometrie e ne fa un sottocampionamento, le tecniche descritte nei vari articoli sono per la maggior parte incentrati sul rendering in real time (grafica raster), quindi si è deciso di seguire la tecnica del Multisampling ma di fare le valutazioni

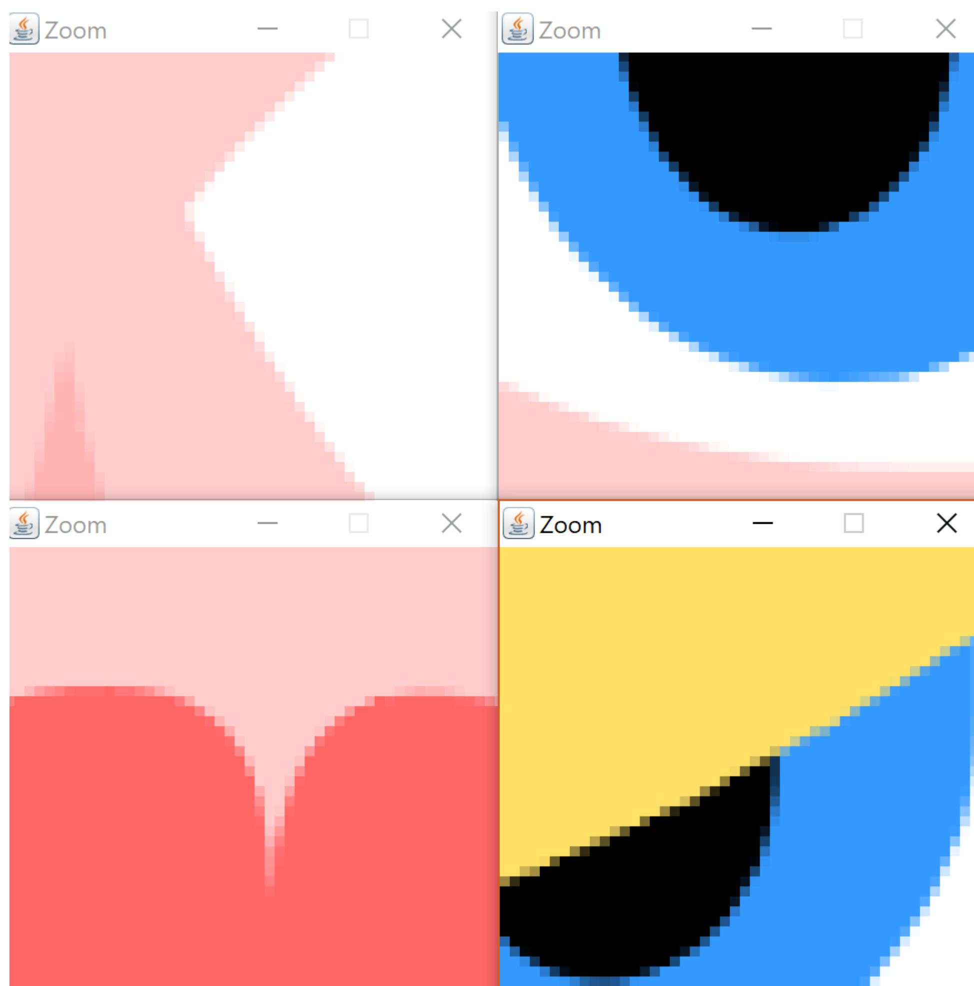

Figura 4.14: MSAA 16x

del campionamento su una geometria generata a risoluzione superiore (per semplicità il nostro campionamento sarà sempre disposto come una griglia quadrata di sub-pixel) così da raggiungere una profondità di dettaglio maggiore nella fase di campionamento (altrimenti la combinazione di colore del campionamento della geometria se fatto sulla stessa solamente scalata ma senza l'aggiunta di dettagli apparirebbe molto sagomata e non sfrutterebbe le potenzialità offerte da una grafica vettoriale).

4.4 Conservative Morphological AntiAliasing (CMAA)

Il Conservative Morphological Antialiasing, rispetto ad altre tecniche ppaa offre una stabilità temporale migliore e si basa su un algoritmo che gestisce solo discontinuità simmetriche (più prudente) per evitare sfocature indesiderate.

Per la fase di implementazione si hanno passaggi di seguito spiegati.

4.4.1 Analisi dell'immagine per discontinuità di colore

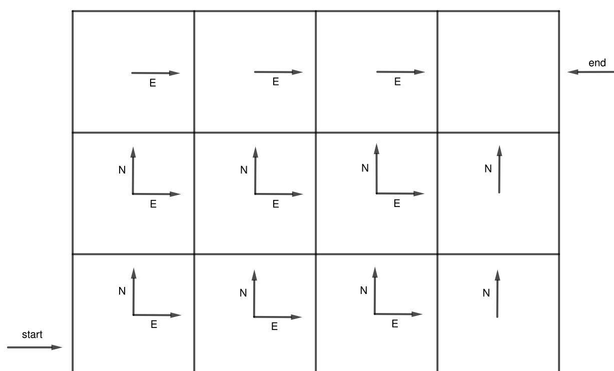

Figura 4.15: Rilevamento bordi per pixel

Memorizziamo i lati in cui avviene questa discontinuità (non esclusivo del CMAA). Per questa prima fase di approvigionamento dei lati in cui avviene discontinuità abbiamo scelto una rappresentazione numerica che ci agevoli nel determinare i pixel coinvolti, rappresentando il nostro framebuffer come una griglia, i punti in cui può accadere discontinuità non è il pixel stesso, ma il bordo. Ed in più il totale dei bordi individuati può essere rappresentato da due bordi per pixel, qualsiasi sia il senso di orientamento con la quale andremo ad individuare essi. Ad esempio il $pixel(0,0)$ può al massimo rappresentare se ha il bordo ad $EST(0.5,0)$ e/o a $NORD(0,0.5)$, questa numerazione è solo

rappresentativa, non bisogna confonderla con la posizione reale del bordo. In questo modo possiamo facilmente ricavarci i bordi adiacenti (o almeno la sua rappresentazione) per poi verificare che faccia parte dell'insieme dei bordi rilevati o meno, per ricavarci i bordi vicini possiamo applicare dei piccoli calcoli all'attuale bordo preso in considerazione, e possiamo muoverci verso i punti cardinali:

$$NORD(lato) = x, y + 1$$

Quando ci si dirige verso nord la destinazione sarà sempre un bordo di tipo orizzontale se il punto di partenza è orizzontale e viceversa.

$$NORD - EST(lato) = x + 0.5, y + 0.5$$

Quando la nostra direzione è inclinata rispetto ai punti cardinali principali, la nostra destinazione sarà orizzontale se il bordo di partenza e verticale e viceversa.

$$EST(lato) = x + 1, y \qquad SUD - EST(lato) = x + 0.5, y - 0.5$$

$$SUD(lato) = x, y - 1 \qquad SUD - OVEST(lato) = x - 0.5, y - 0.5$$

$$OVEST(lato) = x - 1, y \qquad NORD - OVEST(lato) = x - 0.5, y + 0.5$$

Inoltre questa rappresentazione porta con se la semplicità di individuare i pixel coinvolti del bordo corrente, per uno dei bordi ci basta prendere il nostro valore e sommare per una costante piccola ($+0.1$) entrambe (bx, by) e dopo arrotondare i valori ad interi, per l'altro pixel adiacente invece di sommare facciamo la differenza (-0.1).

Per verificare se siamo o meno in presenza di un bordo verticale, controlliamo semplicemente che la parte decimale della x sia maggiore di 0 , se lo è siamo in presenza di un bordo "verticale" altrimenti sarà "orizzontale", dato che il rilevamento dei bordi può e deve aggiungere un tipo solamente di bordo per volta (ad esempio: troviamo discontinuità a nord ed est sullo stesso pixel, vuol dire che andremo ad aggiungere ...$(x, y + .5f)$, $(x + .5, y)$

ogni bordo quindi deve essere aggiunto separatamente). Quando chiamiamo la nostra funzione per il rilevamento dei bordi ci assicuriamo di avere un punto di accesso abbastanza veloce per cercarli, ad esempio noi memorizzeremo una matrice di float con il numero di righe uguale all'altezza del framebuffer e di colonne 1 (inizialmente), ad ogni rilevamento della riga lo ricordiamo e poi creiamo un array da inserire alla posizione della matrice, così da avere le posizione ordinate ed accessibili in maniera precisa (riga per riga). Per rilevare la discontinuità di colore eseguiamo un semplice calcolo sulla distanza tra i colori coinvolti:

$$float\, distance(c1, c2)$$

$$d = |r1 - r2| + |g1 - g2| + |b1 - b2|$$

Il valore d verrà confrontato con una costante c, se $d > c$ allora memorizziamo tale bordo.

4.4.2 Rilevatore delle forme Z e applicazione del nuovo colore

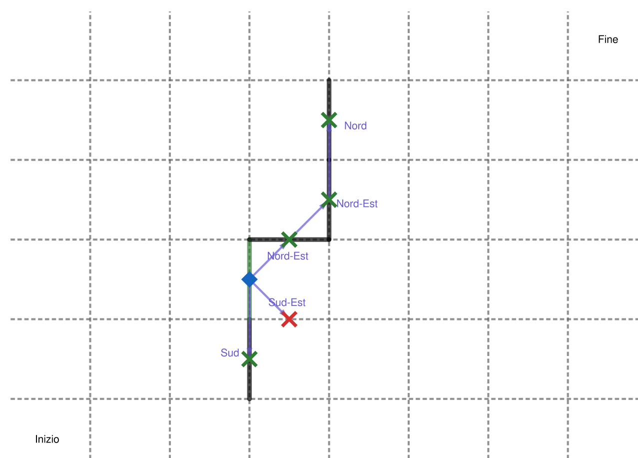

Figura 4.16: Ricerca Z trovata a Nord-Est

Per rilevare le forme possiamo procedere ricercando a NORD-EST e SUD-EST dal bordo attuale (questo è possibile perché cerchiamo una for-

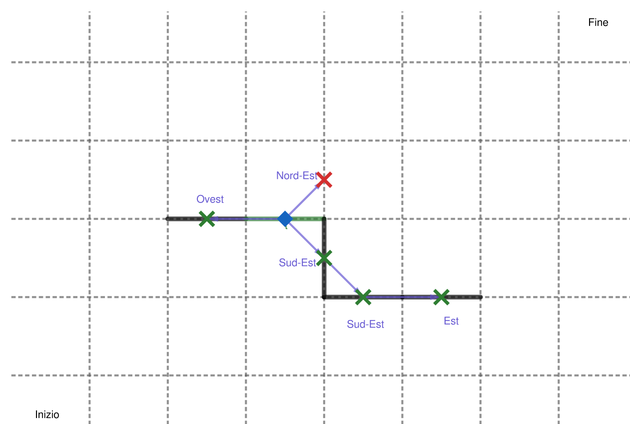

Figura 4.17: Ricerca Z trovata a Sud-Est

ma simmetrica, non importa da quale lato della forma iniziamo, perché sia da un lato che l'altro lato porta alla stessa forma, e per risparmiare quindi ci concentriamo solo sulla ricerca a NE e SE) :

Ricerca a NORD-EST

1. Dal bordo attuale controlliamo se esiste a NordEst
2. Continuiamo a controllare a NordEst, se esiste abbiamo la forma Z
3. Guardiamo l'orientamento della forma trovata (orizzontale o verticale)
4. Verifichiamo per quanto si estende
5. Applichiamo le modifiche al colore su tutta la forma trovata

Ricerca a SUD-EST

1. Dal bordo attuale controlliamo se esiste a SudEst
2. Continuiamo a controllare a SudEst, se esiste abbiamo la forma Z
3. Guardiamo l'orientamento della forma trovata (orizzontale o verticale)
4. Verifichiamo per quanto si estende

5. Applichiamo le modifiche al colore su tutta la forma trovata

Per entrambi le iterazioni se al passo 2 non troviamo il bordo desiderato allora sarà una delle forme semplici descritte nel capitolo 3 sulla parte riguardante il CMAA.

4.4.3 Considerazione finale

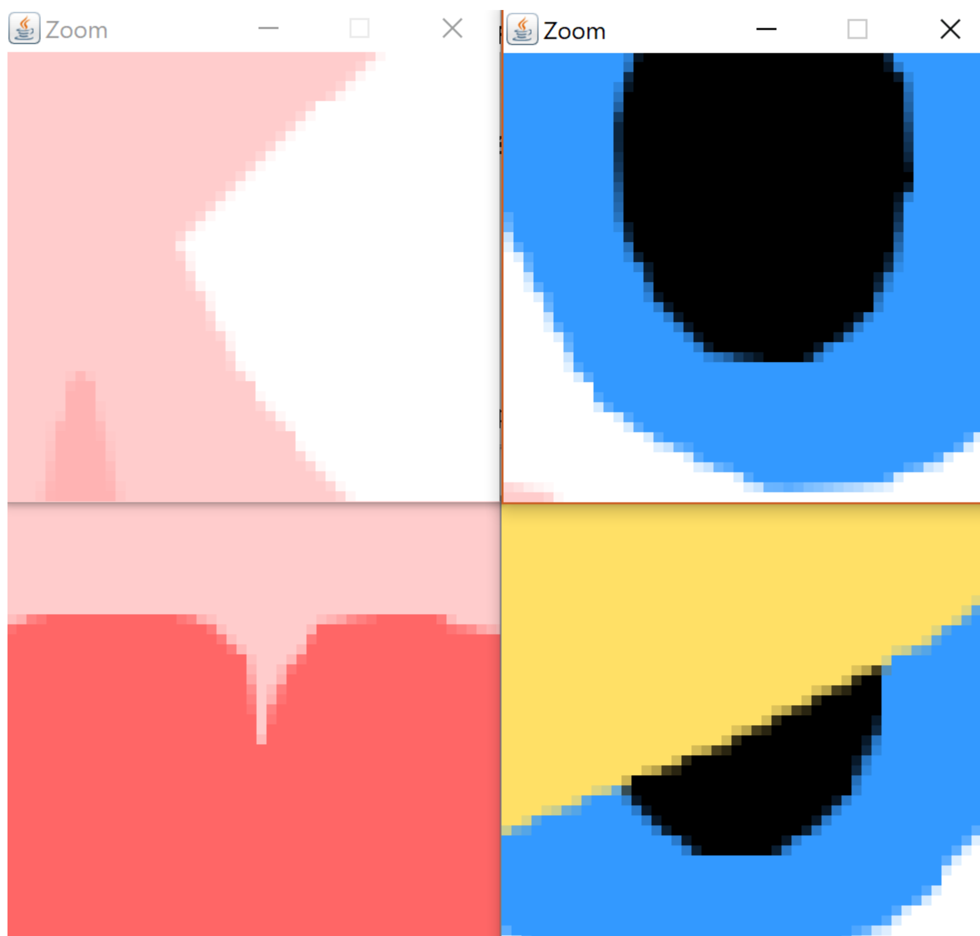

Figura 4.18: CMAA: conservative morphological antialiasing

Il CMAA è molto semplice da implementare e non richiede grandi risorse, può essere usato in applicazioni RealTime ed è applicabile a tutta l'immagine, non si adatta ad una grafica vettoriale, perchè non riesce a rendere fluide

forme molto inclinate. Questo filtro non riesce soprattutto a migliorarne significativamente la qualità perché non abbiamo la reale informazione di quanto colore applicare e soprattutto non sfruttiamo il dettaglio offerto da una grafica vettoriale.

4.5 Confronto AntiAliasing

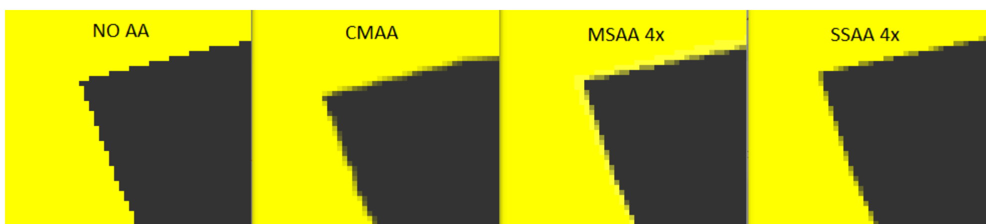

Figura 4.19: Confronto AntiAliasing

Come si può notare dalla figura 4.19 e 4.20 tutte le tecniche messe a confronto cercano di smussare la scalettatura, ma come è facilmente visibile hanno tutti risultati differenti. Il CMAA individua i punti di Aliasing dopo aver rasterizzato tutta l'immagine, ne ricerca le forme ed infine applica le modifiche, risulta di facile implementazione ma non permette di ricavare con esattezza la quantità di colore da modificare in quei determinati pixel, inoltre per forme con grandi inclinazioni l'aliasing è ancora visibile, questa sua pecularità non lo rende appropriato per un immagine vettoriale (che potenzialmente non ha un limite al dettaglio raggiungibile) . MSAA invece

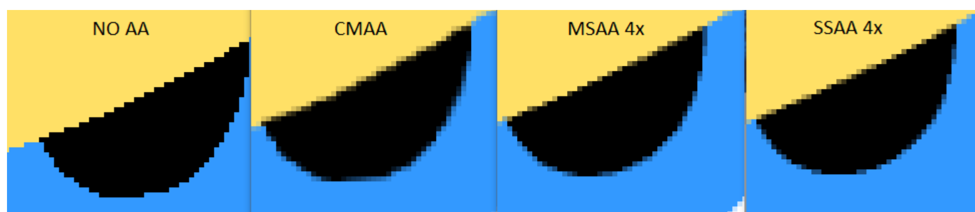

Figura 4.20: Confronto AntiAliasing

permette di elaborare le informazioni con esattezza, confutando quanto quel

poligono copre un determinato pixel, con risultati davvero ottimi (inoltre la qualità dipende dal numero dei campionamenti effettuati su un determinato pixel).

Guardando le figure possiamo osservare anche delle differenza tra il MultiSampling ed il SuperSampling, questo è dovuto alla disposizione dei punti campioni, nel primo sono stati disposti ruotandoli dal loro centro (Rotated Grid Super Sampling), nel secondo invece può essere vista come una griglia ordinata (Orderer Grid Super Sampling). Una griglia ordinata risolve la maggior parte dei problemi di aliasing vicino all'asse orizzontale e verticale, invece ruotando leggermente la disposizione dei punti campione si riesce ad ottenere un leggero incremento qualitativo.

4.6 Offset di Curve

L'offset di una curva (come accennato nel capitolo 2) può essere calcolato tramite:

$$Cd(t) = C(t) + d * N(t)$$

d è la distanza dal punto originale della curva.

Nella realizzazione dell'offset di una curva può in alcuni casi (se non è una curva regolare) intersecare con se stessa (figura 4.21), creando anomalie nei calcoli successivi (ad esempio quando riempiamo il tratto dell'offset della curva dove all'interno si trova un autointersezione, quell'area non sarà riempita correttamente a causa della regola even-odd che usa la Scan Conversion).

Per ovviare a questo inconveniente valutiamo la polilinea offset con se stessa cercando le linee che intersecano, se esiste tale punto allora l'intervallo di linee che si trova tra il punto di intersezione sarà eliminato. Per realizzare un tracciato (sequenza di curve dal quale creare l'offset) come si nota dalla figura 4.22, se non c'è continuità tra una curva e la successiva, allora bisogna calcolarsi la parte mancante da un lato e tagliare la parte che interseca dall'altro (le due azioni descritte sono complementari, se esiste l'uno esite anche l'altro).

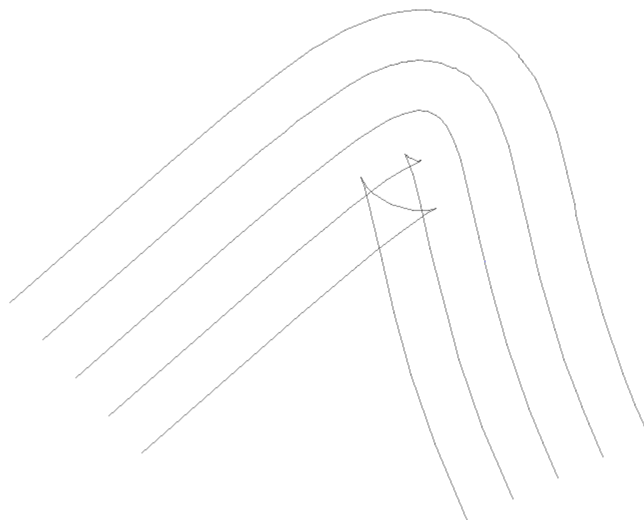

Figura 4.21: AutoIntersezione dell'offset di una curva.

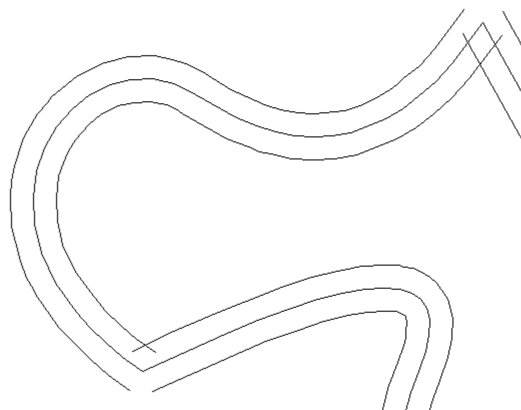

Figura 4.22: Tracciati senza gestire le connessioni tra le curve.

Come si evince dalla figura 4.23 disegnando un cerchio tra una curva e la successiva, possiamo osservare che le linee dell'offset iniziano e finiscono sulla circonferenza, da ciò abbiamo la conferma che l'offset è realizzato nella sua interezza da tutti e due i lati, osservando attentamente si può notare che la parte mancante si trova trasposta dal lato opposto, dal punto di intersezione

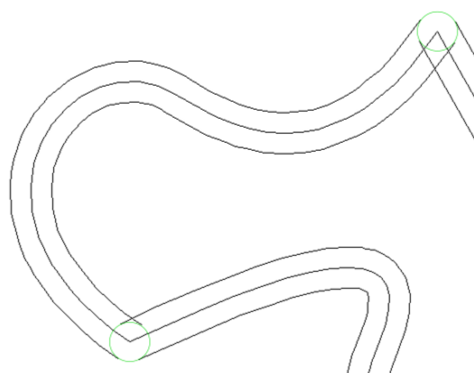

Figura 4.23: L'offset sugli spigoli.

fino alla circonferenza del cerchio.

Preso l'intervallo sottratto dagli offset di curve adiacenti procediamo prima con l'inversione della posizione dei punti, dopo aggiungiamo il punto di intersezione tra i due tagli sottratti dalla curva con la successiva, infine per posizionarli dal lato giusto si effettua una scala con valore -1 dal punto geometrico (la fine o inizio della curva), non c'è bisogno di calcolarsi la parte mancante perché era già presente e in più possiamo utilizzare tali punti per realizzare i tipi di spigoli diversi.

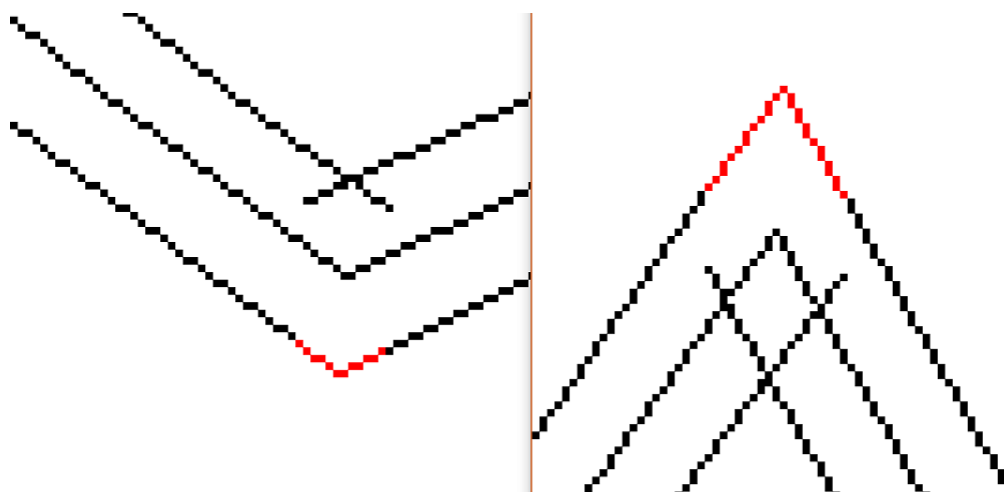

Figura 4.24: Spigolo Vivo

Quello appena descritto viene chiamato “Spigolo Vivo”, per realizzare lo “Spigolo Arrotondato” possiamo usare il punto geometrico come centro, dall’intervallo sottratto usiamo solo l’inizio e la fine (oppure l’ultimo punto dell’offset o il primo del successivo) per poi creare il tratto di cerchio mancante, per lo “Spigolo Tagliato” uniamo semplicemente il punto iniziale e finale.

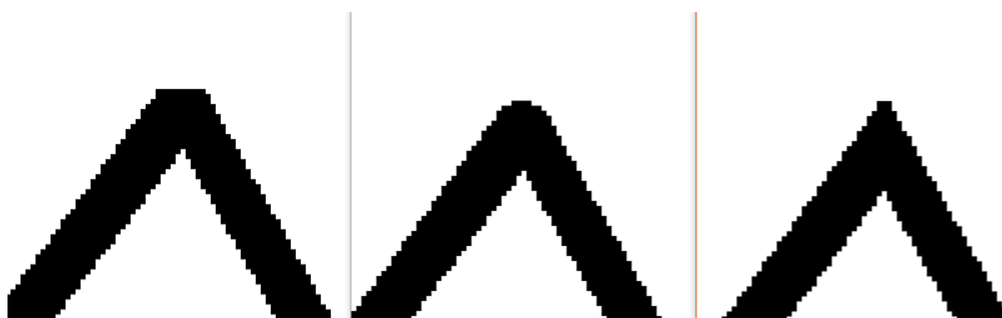

Figura 4.25: Spigoli: sulla sinistra lo “Spigolo Tagliato”, al centro “ Spigolo Arrotondato”, a destra lo “Spigolo Vivo”.

Un percorso può essere chiuso o aperto, se siamo in un percorso chiuso allora i vari tagli e controlli devono essere effettuati anche tra la prima ed ultima curva, se è un percorso aperto allora gli estremi possono essere definiti come “Estremo Arrotondato”, “Estremo Squadrato”, “Estremo Geometrico”.

L’offset di una curva viene tenuta separata dagli altri e soprattutto manteniamo una corrispondenza tra gli offset di sinistra e quelli di destra, quando si aggiunge la parte mancante viene unito con l’offset corrispondente, usando questa metodologia ogni polilinea sarà in corrispondenza uno ad uno e risulta semplice creare il poligono di una curva per poi essere riempito (rasterizzato) tramite la Scan-Converter.

4.7 MSAA nei tracciati

Per eliminare l’aliasing possiamo utilizzare il MultiSampling (MSAA), come descritto nelle sezioni precedenti si procede verificando la copertura dei

punti campioni con il tracciato a risoluzione superiore.

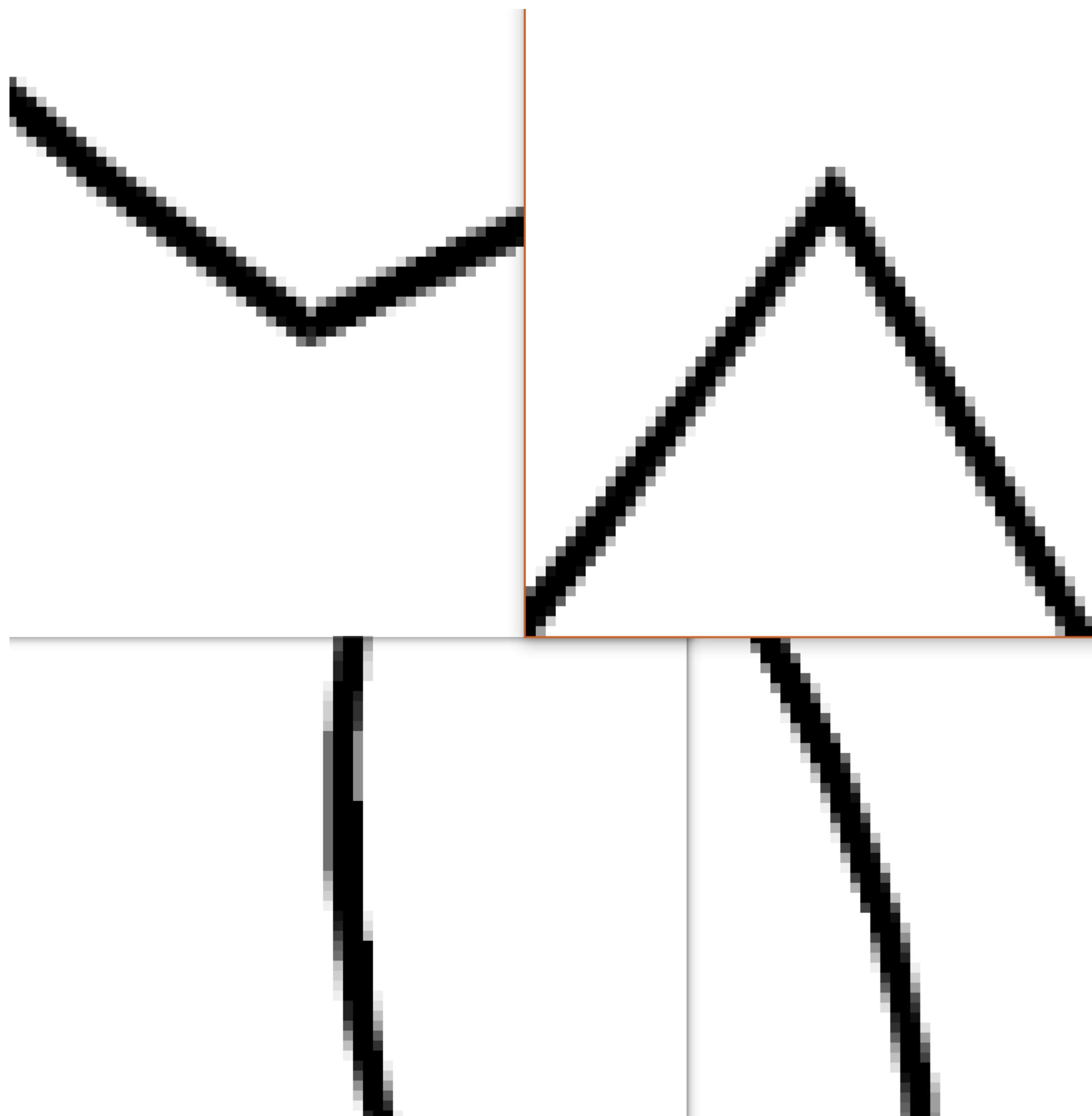

Figura 4.26: Tracciati con MSAA 16x

Per effettuare la fase di campionamento si è scelto di creare l'algoritmo di triangolazione che ci ritorna una sequenza continua dei triangoli che riempiono il tracciato, si procede valutando singolarmente l'offset di una curva alla volta. Abbiamo chiamato il nostro algoritmo " Triangolatore Binario" perché esso si muove sugli offset (che possono essere visti come dei binari).

Una delle prime difficoltà incontrate è nella scelta dei punti degli offset da utilizzare per creare il triangolo, dato che la disposizione dei punti non è equidistante e in una curva molto stretta l'offset esterno avrà molti più punti di quello interno, per cui bisogna stabilire la distanza percorsa (tra l'offset sinistro ed il destro) in base alla curva originale, per valutare quale lato far procedere si è deciso di utilizzare la normale della curva generando un punto offset a sinistra e a destra, da valutare con l'offset creato in precedenza, chi sta dietro ed è il più distante dal punto di test creato ha la precedenza a proseguire, se entrambi i lati (offset) sono davanti al punto generato dalla Normale ($N(t)$) si porta avanti t fino a che almeno uno dei due punti offset (sinistro o destro) sia dietro. L'algoritmo di test chiamato "DistanceNormalBinary" può lavorare solo sulla curva, per superare questo limite dato che nel pezzetto di offset può trovarsi (alla fine) il tratto che era stato precedentemente unito per connettere l'offset esterno al successivo, ho deciso di suddividere in tre fasi l'intero algoritmo di triangolazione:

- Il primo passo di triangolazione lo effettua chi ha più punti.
- I successivi passi vengono determinati dalla funzione di test fino a $t = 1$.
- Successivamente se rimangono ancora punti da triangolare si procede iterando semplicemente per i punti rimasti (si troveranno da un solo lato degli offset)

I triangoli sono aggiunti in sequenza su un unico array e verranno utilizzati per il test di copertura del Multisampling.

Infine quando si ridimensiona un tracciato si deve evitare che lo spessore sia minore di 0.5 altrimenti si può verificare il caso in cui nessun punto campione venga coperto dai triangoli, e quindi non venga disegnato nella maniera corretta.

Un tracciato può essere utilizzato per rasterizzare una linea con antialiasing, l'algoritmo ha ottimi risultati qualitativi perché crea uno spessore con cui effettuare i calcoli, lavora in virgola mobile, quindi è possibile operare calcoli sotto la dimensione del pixel.

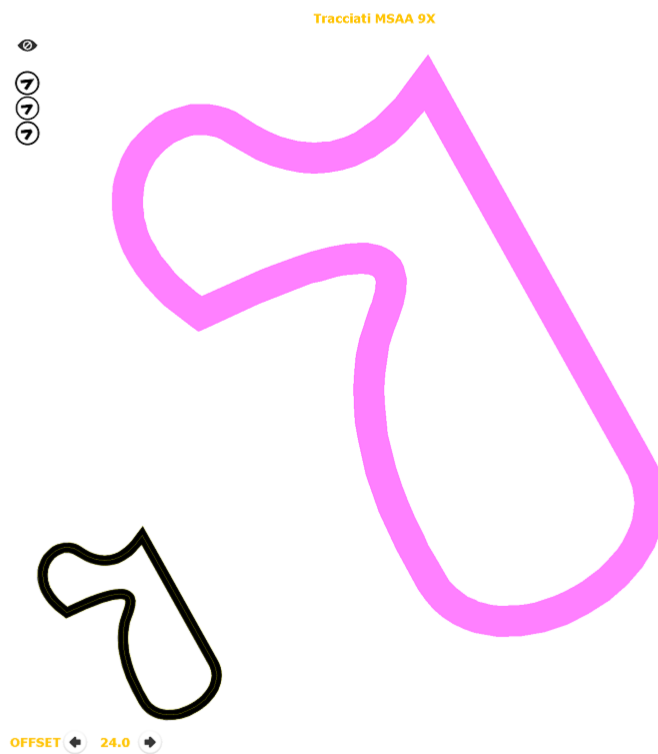

Figura 4.27: Un esempio della triangolazione del tracciato a risoluzione maggiore (viola)

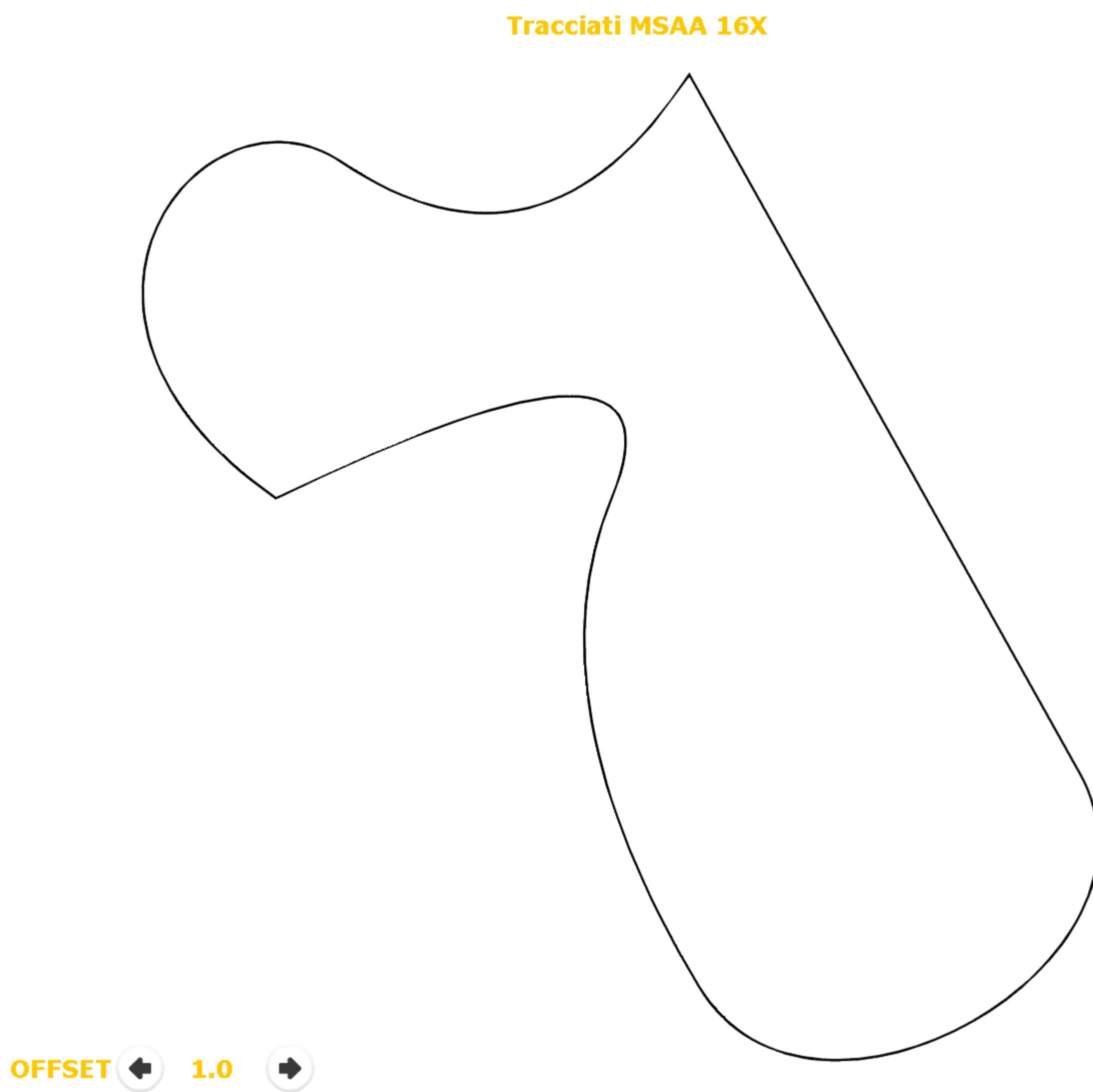

Figura 4.28: Tracciato completo con Antialiasing

Figura 4.29: Utilizzo del tracciato per rasterizzare la polilinea della curva con antialiasing.

Conclusioni

Questo lavoro di tesi si è occupato di Grafica Vettoriale e Rasterizzazione, evidenziando che la parte più complessa della gestione delle immagini vettoriali è la loro rasterizzazione, cioè la necessità di riportare il disegno effettuato su dispositivi raster.

Particolare attenzione è stata rivolta alle tecniche esistenti per rimuovere l'aliasing creato dal processo di "Rasterizzazione" in genere e in particolare nella "Grafica Vettoriale".

Si è osservato come il MSAA sia il migliore tra quelli proposti in letteratura (in quanto a qualità), e come possa essere utilizzato per il "path rendering" (disegno vettoriale di linee e curve).

A tal fine si è realizzato un ambiente di prova (linguaggio JAVA) dove sono state implementate e studiate le tecniche e gli algoritmi descritti nei vari capitoli, confrontandoli e mettendo in evidenzia i loro pregi e difetti.

Bibliografia

[1] Gomez Graphics Vector Conversions, Raster vs Vector, *https://vector-conversions.com/vectorizing/raster_vs_vector.html*

[2] Nvidia Corporation, An Introduction to NV path rendering, Accelerating Vector Graphics for the Mobile Web, *https://developer.nvidia.com/nv-path-rendering* , (June 8 2011)

[3] Alexander V. Reshetov(CA), David Patrick Luebke(VA), Nvidia Corporation, Santa Clara, CA(US), *Infinite Resolution Textures*, (June 7 2018)

[4] Giuseppe Attardi, Anna Bernasconi, Università di pisa, *Fondamenti di Computer Graphics*, (1996/1997)

[5] Giulio Casciola, Dipartimento di Matematica Università di Bologna, *Matematica e Informatica: dietro le quinte della grafica al calcolatore*, (2008/2009)

[6] Gershon Elber, In-Kwon Lee, and Myung-Soo Kim, *Comparing Offset Curve Approximation Methods*, (1997)

[7] Alvy Ray Smith July 17, *A pixel is Not a Little Square*, (1995)

[8] Xiaolin Wu, Departement of Computer science University of Western Ontario, *An Efficent Antialiasing Technique*, (1991)

[9] Kristof Beets, Dave Barron, Beyond3D, *Super-sampling Anti-aliasing Analyzed*

[10] Matt Pettineo, *A Quick Overview of MSAA, https://mynameismjp.wordpress.com/2012/10/24/msaa-overview/*

[11] ACM SIGGRAPH Course, *Filtering Approaches for Real-Time Anti-Aliasing, http://iryoku.com/aacourse/* , (2011)

[12] Alexander Reshetov, Intel Labs, *Morphological Antialiasing* , (2009)

[13] By Filip Strugar (Intel), Leigh D. (Intel), *Conservative Morphological Anti-Aliasing (CMAA)*, (March 18, 2014)

Ringraziamenti

Alla mia famiglia che ha sempre creduto in me.

Tiziana la mia ragazza, per il sostegno e la pazienza per tutto il periodo di studio.

Tutti i miei amici che mi sono sempre stati vicini.

Al professore Giulio Casciola per la guida e la pazienza avuta durante il lavoro di tesi.