סיכום החומר בגרפיקה

רזולוציה של תמונה – לכל התקן המציג תמונות יש 3 פרמטרים הקובעים את הרזולוציה של התמונה המוצגת:

- **עומק הצבע Intensity resolution** – לפי כמות הביטים שבאמצעותם מיוצג פיקסל אנו יודעים כמה הצבע "מדוייק" (למשל: 8 ביט, 16 ביט, 32 ביט וכו').

- **רזולוציה מרחבית Spatial resolution** – בתמונה יש מספר פיקסלים השווה למספר הפיקסלים לרוחב כפול מספר הפיקסלים בגובה.

- **רזולוציית רענון Temporal resolution** – מספר הפעמים בהם המסך מתעדכן (קצב רענון). רענון המסכים יכול להתבצע ב-2 שיטות: ע"י עדכון המסך פיקסל אחר פיקסל לפי שורות או באמצעות עדכון חוצץ (Buffer) והחלפה בין החוצץ המוצג.

אור נראה – ניתן לתאר צבע של אור ע"י 3 מאפיינים:

- **גוון (Hue)** – מתאר את התדר השולט (הPeak הגבוה של תדר הצבע).
- **Saturation** – מתאר כמה הצבע "עשיר" (היחס בין הכי גבוה לאחרים).
- **Lightness** - כמה הצבע בהיר (השטח שמתח לתדר).

הקולטנים בעין האנושית הרבה יותר רגישים לצבע הירוק והאדום.

ניתן לייצג צבעים בתלת מימד ע"פ מספר שיטות:
- **RGB** – באמצעות נקודה בקובייה תלת מימדית.
- **Polar Color** – נקודה בתוך צילנדר תלת מימדי (הגובה מתאר בהירות, המרחק מהמרכז מתאר "עושר" של הצבע והזווית בהיקף מתארת את הגוון).
- **צבעים נגדיים** – נקודה בקובייה. כל ציר מסמל זוג של צבעים נגדיים.

מודלים של צבע

- **מודל לנארי** - בRGB אנו מתחילים מ-0 ומוסיפים עד 1 (בדומה למסכים המתחילים בשחור ומגיעים ללבן). בCMYK מתחילים מ-1 ויורדים ל-0 (בדומה למדפסות המתחילות בלבן ומסיימות בשחור).

- **Artistic View** – (HSB/HSV) מנסה לדמות את תפיסתי אך במודל פשוט יותר.

- **סטנדרטי** – CIE בשנת 1931 עשו ניסויים והגיעו לסקאלה של צבעים הנראים ע"י בני אדם. משתמשים בסקלה זו כדי להשוואות בין רכיבים שונים במציגים צבעים (ע"פ תת הסקאלה של הצבעים אותה הם יכולים להציג). כך ניתן לכייל מדפסות והתקנים אחרים (למשל). לכל מכשיר כזה יש 3 נקודות היוצרות את משולש הגמא המתאר את מרחב הצבעים אותו הוא יכול להציג. הסקאלה נורמלה להיות בין 0 ל-1.

- **תפיסתי (Perceptual)** – RGB הוא מודל לא טוב לתיאור צבעים כי הוא לנארי ואנו קולטים צבעים בצורה לא ליניארית (למשל מבדילים יותר טוב בשינויים של אדום וירוק). לכן פותח המודל הזה. המיוחד במודל הוא בכך שמרחק בין נקודות במרחב הצבעים במודל מסמל יותר טוב את ההבדל שהעין האנושית תופסת. כלומר, 2 נקודות הרחוקות באותה מידה יסמנו צבעים שנתפסים שונים במידה שווה (לעומת RGB). המרחק במודל זה אינו לנארי (הבדל בין מרחק 5 למרחק 10 אינו בהכרח פי 2).

- **Opponent** – משתמשים בטלוויזיות.

Aliasing – מצב בו יש פונקציה בתדירות גבוהה ואנו דוגמים בתדירות נמוכה מדי וע"י כך מאבדים מידע מהתמונה.

כיצד יודעים כמה מהר סיגנל משתנה? באמצעות תדר.
התמונה שאנו מנסים לדגום נמצאת בדומיין המרחבי. ניתן להעביר כל סיגנל מהדומיין המרחבי אל תדר (באמצעות מעבר פורייה). כל תדר בדומיין התדרים ניתן לייצג באמצעות פונקציות מחזוריות (גם אם הוא עצמו לא מחזורי). כדי להתקרב לפונקציה המקורית, אנו מבצעים הוספה של פונקציות עם תדר הולך וגדל בשיטת פורייה.

AntiAliasing – ניתן באמצעות דגימה בקצב גבוה יותר (לא תמיד אפשרי ולא תמיד פותר את הבעיה) או ע"י הפעלת פילטר כדי ליצור סיגנל עם תדירות מוגבלת, Bandlimited (מחליף את הAliasing בBlur).

Bandlimited – השיטה האידיאלית להגבלת התדירות היא באמצעות נרמול sin.

Image Processing

התאמת בהירות – כדי לשנות בהירות של תמונה מבצעים טרנספורמציה לנארית על כל פיקסל (תוך שמירה על הערכים בטווח המותר).

התאמת ניגודיות – מחשבים ממוצע צבע עם משקולות מתאימים לכל צבע (נקבע במחקר) ומגדילים או מקטינים את המרחק של כל פיקסל מהממוצע הזה (הגדלה שווה להוספה והקטנה שווה להורדה).

פילטרים ליניאריים – מגדירים פילטר ע"פ הפיקסלים שמסביב לפיקסל הנוכחי. מבצעים באמצעות המשקולות שניתנות לכל פיקסל ממוצע משכולל והצבע המתקבל הוא הצבע של הפיקסל (סכום המטריצה המגדירה צריך להיות 1).

- **Blur** - נגדיר מטריצה ריבועית עם משקל בכל אחד מהתאים. ניתן לתת משקל אחיד לכל פיקסל או משקל משתנה. בשיטה הגאוסיאנית נותנים משקל גבוה יותר לפיקסל הקרוב למרכז. עבודה עם גרעין גאוסינאי טוב יותר כיוון שהיא נותנת תוצאות חלקות יותר (שינוי דרסטי בפיקסל אחד פחות משפיע מאשר בשיטה האחידה). זה נובע מהעובדה שתדר של פילטר גאוסיאני הוא גאוסיאני ולעומתו בפילטר האחיד יש השפעה גבוהה לכל פיקסל.

- **זיהוי גבולות Edge Detection** – בשיטה זו מגדירים את הפיקסל המרכזי (במטריצה nxn) ערך של n^2-1 ושאר הפיקסלים עם ערך 1-. כך, אם הסביבה קרובה לערך הפיקסל, מתקבל 0, אחרת מתקבלים ערכים גבוהים.

- **Sharpen** – בצורה דומה לזיהוי אך הפעם הפיקסל המרכזי מקבל ערך של n^2. בעצם אנו לוקחים את התמונה המקורית ו"מוסיפים" לה את הערכים של הEdge Detection.

- **Emboss** – מציג גרדיאנטים בכיוון מסוים.

פלטרים לא ליניאריים – מבצעים כל פעולה לא ליניארית על הפיקסלים הסובבים את הפיקסל הנוכחי. למשל במקום לבצע ממוצע (עבור Blur) בוחרים את החציון. הסיבה שמתקבלות תוצאות טובות יותר הינה כיוון שלחציון נקודת שבירה גדולה יותר מהממוצע. החיסרון שיותר קשה לחשב חציון ממוצע.

Scaling – כדי לבצע הקטנה/הגדלה צריך לעיתים פיקסלים שונים שלא קיימים בתמונה המקורית. לכן, מקרבים את הפונקציה המקורית באמצעות המידע שיש לנו ואז מבצעים דגימה מחדש.

Image Resampling – אנו רוצים לדגום נקודות שאין לנו. לכן עלינו למצוא את ערכם באמצעות הסביבה במספר שיטות.

- **שיטת המשולש** – נותנת לכל נקודה שמרחקה קטן מ-w מהנק' שאנו דוגמים את המשקל 1- d/w (כש-d זה המרחק של הנקודה הידועה מהנקודה שאנו רוצים לדגום). כשאנו מסתכלים

רק על 4 הפיקסלים הקרובים אלינו אנו מבצעים ממוצע משוקלל של 2 הזוגות (2 הפיקסלים העלווניים והשניים התחתונים) ולתוצאות מבצעים גם ממוצע משוקלל.

- **השיטה הגאוסנית** – נותנים משקל ע"פ פונקצית גאוס.
- **Point Resample** – לוקחים את הנקודה הקרובה ביותר.

Image Warping – מבצע מיפוי של הנקודות מהתמונה הישנה לתמונה החדשה. ניתן לבצע את הדרכים בין 2 שיטות: מיפוי כל פיקסל בתמונה הישנה לתמונה החדשה (ואז חישוב הסביבה על פיו) או מיפוי כל פיקסל חדש לתמונה הישנה (וחישוב הערך שלה ע"פ הסביבה בתמונה הישנה).

מיפוי באמצעות נקודות – מגדירים 3 נקודות בתמונה המקורית ומיקומם בתמונה החדשה. עתה, בתוך המשולש, ניתן לחשב את הערך של כל נקודה באמצעות הערכים של קודקודי המשולש (יש נוסחאות). כדי לחשב את הערכים של כל הנקודות בתמונה החדשה, מבצעים טריאנגולציה בתמונה המקורית, מעבירים את המשולשים למצב החדש ואז מצבעים חישוב של כל פיקסל (כל פיקסל נמצא עתה בתוך משולש).

Quantization – רוצים להוריד את עומק הצבע בתמונה (למשל להעביר מ16 ביט ל-8 ביט). יש מספר שיטות לבצע זאת:

- **אחידה** – מעגלים את הצבע לצבע הכי קרוב הקיים (יוצר כמו מדרגות בצבע מצבע "חלק").

כדי **להקטין את האפקט** הרע של ה-Quantization, יש מספר שיטות:

- **Dithering** – פיזור השגיאה על פני מספר פיקסלים.
  - **הוספת רעש אקראי** – לוקחים את המידע המקורי ומוסיפים לו "רעש" אקראי.
  - **Ordered Dither** - קיימות מטריצות מוגדרות של Bayer המגדירות כמה שגיאה כל פיקסל מקבל. השאיפה היא לתת לכל 2 פיקסלים צמודים רעש כמה שיותר שונה.
  - **Error Diffusion Dither** – מתחילים מקצה שמאל עליון. מבצעים עיגול ואז את השגיאה שנוצרה (כתוצאה מהעיגול) מוסיפים לערכים של הפיקסלים הצמודים מלמטה ומימין ע"פ חלוקה משוקללת שנקבעה. כך ממשיכים הלאה עד הקצה הימיני התחתון.
- **Halftoning** – שיטה להורדת סקאלה של צבעים. למשל בדפוס אנחנו לא יכולים להציג את כל הגוונים אבל אנחנו יכולים ליצור נקודות מאוד קטנות. ניתן להדפיס נקודה גדולה יותר ככל שהצבע יותר Intense. בנוסף, ניתן לחלק כל פיקסל למספר חלקים ולמלא אותו במספר נקודות בהתאם לעומק הצבע. אם חילקנו פיקסל ל-nxn חלקים, יצרנו n^2+1 רמות עומק.

טרנספורמציות מודלים

קיימים 3 סוגים של טרנספורמציות:

- **Rigit** – משמר מרחקים. רק משנה מיקום ולא משנה את הצורה (הזזה וסיבוב).
- **Similarity** – הצורות נשארות דומות (הזוויות נשמרות) (הזזה, סיבוב, הגדלה/הקטנה אחידה).
- **Affine** – משמרות קווים מקבילים (הזזה, סיבוב, הגדלה/הקטנה, ו-Shear).

אנו יכולים לבצע הרכבה של טרנספורמציות באמצעות מכפלת מטריצות (נוח ויעיל כדי לייצג רצף של טרנספורמציות).

במטריצות מגודל 2x2 (כשגודל הוקטור הוא 2) ניתן לייצג רק טרנס' ליניאריות. למשל, ניתן לייצג סיבוב, Shear, הגדלה/הקטנה, סיבוב סביב הצירים וכו' אך לא ניתן לייצג הזזה. נשים לב שבמקרים אלו (0,0) נשאר ב-(0,0).

כדי לפתור את הבעיה, אנו מוסיפים מימד נוסף למטריצה (1 באלכסון ו-0 באיברים החדשים בשורה התחתונה) ומוסיפים קואורדינאטה לוקטור.
נשים לב שגם זה לא מספיק לנו עבור טרנס' בהם קווים מקבילים לא נשארים כך (למשל פסי רכבת במרחב). כדי לפתור בעיה זו נחליף את הערכים של המטריצה בשורה האחרונה מאפסים למספרים כלשהם.
צריך לזכור שבעקבות כל התוספות הללו, w לא תמיד נשמר להיות 1 בווקטור התוצאה של המכפלה ולכן יש לנרמל לפיו את הווקטור.
חשוב: מכפלת מטריצות אינה קומוטטיבית ולכן סדר ההכפלה משנה. ככל שאנו מתרחקים מהנקודה, הטרנס' הופכת להיות יותר גלובלית ופחות לוקלית.

כדי לבצע טרנס' בתלת מימד נוסיף מימד נוסף למטריצה. השימוש בטרנס' נשאר דומה מאוד עדיין.

היררכיית הטרנספורמציות – כדי לצייר גוף המורכב ממספר חלקים (ליצן למשל) נגדיר לכל חלק מטריצה המביאה אותה ל"הורה" שלה. למשל, לכף יד תהיה מטריצה המעבירה אותה ביחס לאמה ואת האמה ביחס לזרוע וכו').

Rendering
מערכת של רינדור בתלת מימד צריכה לדאוג למספר מאפיינים (ע"פ חשיבות):

- **מצלמה (Pin-Hole)** - כל הקרניים מגיעות לנקודה אחת, ללא עיוות, הצבע בנקודה נקבע לפי התאורה המגיעה מהנקודה לחיישן, כל העולם נמצא בפוקוס. המצלמה כוללת את הפרמטרים הבאים: מיקום, אוריינטציה כלפי מעלה וקדימה, נקודה עליה מסתכלים/נורמל של המשטח ומידת הפתיחה.

- **מה אנחנו רואים** – אנו מחליטים על הצבע של כל פיקסל לפי הקרניים הפוגעות בו. אנו נעבוד בשיטת **Ray Casting** ונשגר קניים מהמצלמה דרך כל פיקסל.

- **תאורה ושיקוף** (רמת החזרת האור ממשטחים מסוימים) – קיימים סוגים רבים של תאורה ושל משטחים. סוג המשטח מגדיר כיצד הוא מגיב לכל סוג של תאורה. לתאורה קיים פרמטר המגדיר את היחלשותו באטמוספרה (ככל שהוא רחוק יותר מהמקור הוא נחלש). פרמטרים נוספים של התאורה:

  Phong Reflectance – הוא מודל אנליטי לחישוב תאורה. מורכב מ-

  - **Diffuse** – מדמה משטח מט. האור מתפזר באופן שווה לכל הכיוונים. תלוי בכיוון האור למשטח (N·L).
  - **Ambient** – צבע רך שנועד לדמות תאורה גלובלית.
  - **Specular** – מדמה משטחים מבריקים (הנקודות הלבנות של הברק). תלוי בכיוון הצופה והאור $(V \cdot R)^n$.
  - **Emission** – מדמה את מידת פליטת התאורה של האובייקט (ככל שיותר גדול האובייקט נראה לבן יותר).

- **צללים**

- **תאורה לא ישירה** – מגיעה משיקוף של משטחים. כלומר, האור שהם משקפים החוצה.

- **דגימה**

- בדיקה האם נקודה היא בתוך ספרה, משולש .

האצת בדיקת חיתוך קרן

- **Bounding Volume** – מקיפים כל אובייקט בקופסה (יותר קל לבדוק). אם זה לא פוגע בקופסה, זה בטוח לא פוגע בצורה. בנוסף יוצרים היררכיה של הסצנה לקופסאות. כלומר, כל אובייקט מוקף בקופסה. לאחר מכן, כל מספר אובייקטים מוקפים בקופסה מכילה וכו'.

- **Uniform Grid** – בונים רשת אחידה על כל הסצנה. בכל קובייה אליה נכנס נבדוק אלו אובייקטים נמצאים בה ורק להם נבדוק חיתוך. אם יש חיתוך נעצור. אחרת נמשיך הלאה. היתרון שזה מהיר. החיסרון הוא שאם לא נבחר רשת עם צפיפות חלוקה מתאימה, השיפור לא יהיה טוב

- **Octree** – אנו מחלקים את המרחב לקוביות. בכל שלב אנו מחלקים את המרחב כל עוד יש בקובייה יותר מאובייקט אחד. כאן מעבר התאים יותר מסובך לעומת החלוקה האחידה מצד שני יש פחות קופסאות. מומלץ בעת פיצול לפצל גם את הקופסאות הצמודות, כך בעת חיפוש נצטרך לעלות או לרדת רק רמה אחת.

- **BSP** – מחלקים בצורה רקורסיבית את המרחב באמצעות מישורים. נוצרת היררכיה. בעת החיפוש בעץ בודקים האם הנקודה מעל/מתחת למישור ולפי התשובה מתקדמים.

- **אחרים** – ברוב המקרים, 2 פיקסלים קרובים עשויים לפגוע באותו אובייקט. לכן, תחילה נבדוק אם אנו פוגעים באובייקט שבו פגע הפיקסל השכן. אם כן, קיבלנו חסם מקסימלי.

תאורה ישירה

מידול מקורות אור

מודל אמפירי – זהו מודל אידיאלי ששומר עבור כל נקודה מהי עוצמת האור המגיעה מגיעה אליה ממקורות התאורה ע"פ מדידות אמיתיות (קשה לבצע את כל המדידות ודורש המון מקום אחסון של המידע).

ב-**OpenGL** יש מודלים פשוטים למקורות תאורה:

- **Point Light** – מנורה במרחב הפולטת אור באופן שווה לכל הכיוונים. כוללת את הפרמטרים: עוצמה (intensity), מיקום ומקדמי דעיכה (כמה האור נחלש עם המרחק $I_L=I_0/(k_c+k_1d+k_qd^2)$, כש-d זה המרחק).

- **Directional Light** – מדמה מקור אור כמו שמש. אינו נחלש עם הזמן. מכיל את העוצמה (intensity) וכיוון.

- **Spot Light** – בדומה ל-Point Light אך מכיל גם וקטור כיוון. בכיוון זה האור הינו הכי חזר וככל שמתרחקים לצדדים הוא נחלש. חישוב עוצמת האור ($I_L=I_0(D*L)/(k_c+k_1d+k_qd^2)$ (כש-L זה וקטור ממקור האור אל הנקודה ו-D זה הכיוון של התאורה. כאשר D ו-L באותו כיוון, הזווית ביניהם היא 0 ולכן קוסינוס הזווית הוא 1 ומתקבלת התאורה החזקה ביותר).

מידול משטחים

מודל אמפירי – בדומה למקורות האור.

ב-**OpenGL** יש מודלים פשוט המתבסס על המודל של פונג.

- **Diffuse** – החלק במשטח שמפזר את האור בצורה שווה לכל הכיוונים. כמות האור (המתפזרת בצורה שווה) נקבעת ע"פ זווית הפגיעה של האור במשטח (מודל Lambertian).

- **Specular**– האור פוגע בנקודה מסיימת ואנו מחשבים את וקטור השיקוף R ביחס לנורמל. ככל שהזווית בין המיקום שלנו לווקטור הזה גדולה יותר, כמות התאורה המוחזרת קטנה. **המקדם של פונג** אומר כמה משטח מסוים מבריק.

- **Emission**– מדמה אור היוצא מהאובייקט עצמו. משתמשים במידע לצורך חישוב תאורה גלובלית.

- **Ambient**– מדמה החזרת אור של כל התאורה הלא ישירה.

תאורה גלובלית

צללים – כדי לחשב צללים, אנו מוסיפים מקדם S בחישוב של כל מקור תאורה. מנקודת הפגיעה אנו יורים קרן לכל מקורות התאורה. מקור תאורה שמוסתר ע"י אובייקט, יגרום למקדם להיות 0 ולכן לא נתחשב בו בעת חישוב צבע הנקודה.

נוסחת חישוב התאורה

$$I = I_E + K_A I_A + \sum_L (K_D(N \cdot L) + K_S(V \cdot R)^n) S_L I_L + K_R I_R + K_T I_T$$

- I_E: ה-Emission שנפלט מהאובייקט
- $K_A I_A$: כמות ה-Ambient בסצנה והאחוז שנקלט ע"י האובייקט
- $K_D(N \cdot L)$: כמות תאורת ה-Diffuse של המשטח כתלות בזווית הפגיעה של התאורה
- $K_S(V \cdot R)^n$: כמות תאורת ה-Spucular של המשטח כתלות בזווית של הצופה עם שיקוף וקטור הפגיעה של התאורה
- S_L: עומת מקור התאורה ומקדם המסמן אם יש אובייקט שמסתיר אותו
- $K_R I_R$: במקרה של Ray Tracing רקורסיבי מחשבים את הצבע בנקודת הפגיעה של הקרן החדשה ומכפילים אותו במקדם השבירה/ מקדם השקיפות K
- $K_T I_T$

Anti-Aliasing – כדי לפתור בעיה של Aliasing, במקום לדגום נקודה אחת בכל פיקסל, נוכל לשלוח מספר קרניים מכל פיקסל ולבצע ממוצע של הצבעים שקיבלנו מכל קרן.

- **דגימה אחידה** - כל עוד הדגימה קבועה, לא משנה איך אנו דוגמים (בתוך הפיקסל), העיקר שהמרווחים יהיו שווים (בכל הפיקסלים יחד). דוגמה שלא במרווחים אחידים תיתן לנקודות מסוימות משקל גבוה יותר.

- **Adaptive Super-Sampling** – במקומות בהם יש שינויים גדולים בצבע, נבצע חלוקה לדגימה יותר אינטנסיבית.

- **Stochastic Sampling** – בעיות כמו של מוייר (שנוצרים גלים) לא נפתרות בדגימות אחידות. כדי להתגבר על אנו מחלקים את הפיקסל ל-4 ובכל חלק בוחרים נקודה אקראית דרכה יורים את הקרן (מוסיפים רעש לצורת הדגימה).

פרמטיזציה – באופן כללי, כדי לבצע פרמטיזציה, מוצאים ערכי (u,v) של הנקודה ביחס לאובייקט (הערכים שלהם נעים בין 0 ל-1). משתמשים בערכי uv אלו כדי לקבל את הפיקסל מתוך התמונה אותה רוצים "להלביש" על האובייקט.

תאורה

כאשר משתמשים במקורות תאורה פשוטים, אנו מקבלים צללים פשוטים וחדים. יש אפשרות לפתור בעיה זו באמצעות Area Light.

Umbra – החלק בו "הצל" שמטיל אובייקט הוא מלא (לא מגיע שום אור ממקור התאורה הספציפי).
Penumbra – אזור בו חלק מהאור של מקור התאורה מגיע וחלק ממנו נחסם ע"י אובייקט.

יש 2 מימושים אפשריים ל-Area Light:

- **המימוש הפשוט** – מבוצע ע"י שימוש במספר point lights אשר לכל אחד מהם חלק יחסי בעוצמה הכללית של מקור התאורה. כלומר, אם העוצמה היא I וה-Area Light מורכב מ-n Point Lights אז עוצמת כל אחד היא I/n. חסרונות השיטה: מאוד איטית ובמידה ורזולוציית התמונה אינה טובה, יהיו ארטיפקטים בצל.

- **Monte-Carlo** – מידול מקום התאורה הוא באמצעות ספיר. העוצמה החזקה ביותר של התאורה היא באמצע וככל שמתרחקים מהמרכז העוצמה יורדת. כדי לדעת איזה עוצמת תאורה מתקבלת, יורים n קרניים אקראיות אל הספיר ומבצעים ממוצע על הערך שקיבלנו. חסרות השיטה: התוצאה שלה בממוצע נכונה אך לא תמיד.

לחישוב הרנדומיות ניתן להשתמש גם ב-**Las Vegas**. שיטת לאס וגאס מבטיחה תמיד את התוצאה הנכונה אך הזמן שנדרש לביצוע השיטה מובטח רק בתוחלת (במונטה קרלו הזמן מובטח תמיד אך נכונות התוצאה מובטחת רק בתוחלת).

Path Tracing - שימוש באלגוריתם מונטה קלרו בדומה ל-Ray Tracing – במקום להשתמש בחישוב הפגיעה של הקרן, כדי לירות קרן נוספת, אנו מבצעים יריה של קרניים אקראית מנקודת הפגיעה. בנקודות בהן הקרניים פוגעות יורים עוד קרניים אקראית וכך הלאה עד רמת רקורסיה מסוימת. לבסוף, מוצאים את הצבע של כל הנקודות שפגענו בהם ומצעים שיכלול כדי לקבל את הצבע של הנקודה המקורית.

Pipeline

ב-Ray Tracing היו לנו הרבה אפשרויות (כמו צללים רכים, השתקפויות, צללים חדים וכו'). הבעיה היא שלוקח המון זמן לבנות תמונה, צריך להחזיק כל הזמן את כל העולם בזיכרון, וקשה ליצור תמונות עם השליית פוקוס/Motion Blur. לכן, צריך פיתרון אחר.

Rasterization Based Rendering

יצירת תהליך ליניארי בו ניקח סצנה המורכבת מפרימיטיביים שלאחר פעולה מסוימת עליהם נוכל להפוך אותה לתמונה המבוססת על פיקסלים. אנו משתמשים בשיטת ה-Pipeline כדי לצייר עולם תלת מימדי כתמונה דו ממדית. ה-Pipeline כולל את השלבים הבאים (לפי הסדר):

- **Modeling Transformation** – מעבירים את כל האובייקטים מראשית הצירים למיקום שלו בעולם התלת מימדי שלנו.

- **תאורה** – בהתאם למקורות אור והשתקפויות.

- **Viewing Transformation** – הופכים את ראשית הצירים להיות מיקום המצלמה שלנו. וקטור מעלה הופך להיות ציר y, וקטור ימינה הופך להיות ציר x ווקטור <u>אחורה</u> להיות ציר z. כדי למצוא את מטריצת המעבר T מקואורדינאטות עולם למצלמה, אנו משתמשים במידע על המצלמה כדי למצוא את T (בעיקרון יותר קל למצוא את המטריצה ההופכית לT וממנה לגזור את T). T^{-1} מורכבת מעמודת וקטור ימינה, למעלה, אחורה ומיקום המצלמה.

- **Projection Transformation** – מעבירים את האובייקטים לקואורדינאטות דו מימד. בגדול יש 2 סוגים של טרנס' הטלה: מקבילי ופרספקטיבי. תחת כל סוג כזה קיימים סוגים נוספים. בהטלה מקבילית, כל הנקודות מוטלות באותו כיוון על המסך (**DOP**). הטלה מקבילית נוחה למימוש אך קשה לתפוס באמצעותה עומק. האזור הנקלט בשיטה זו הינו בצורת תיבה. הטלה פרספקטיבית היא יותר פוטו-ראליסטית, בשיטה זו יורים את הקווים לכיוון העין (**COP**) ולא בצורה מקבילית. בנוסף, האזור שנקלט בשיטה זו הוא כמו פירמידה קטומה.

- **Clipping**– אין צורך לצייר אובייקטים שנמצאים מחוץ לחלון הצפייה שלנו. לכן, מבצעים חיתוך של העולם. <u>עבור קווים</u>: מחלקים את העולם ל-9 אזורים בהתאם לחלון. ממספרים כל

מלבן בצורה בינארית כשהחלון הוא 0000. מבצעים בדיקת OR. אם קיבלנו 0, 2 הנקודות בחלון. אחרת מבצעים בדיקת AND. אם לא קיבלנו 0, הקו לא בחלון. אם כן קיבלנו 0, לפחות נקודה אחת מחוץ לחלון, מוצאים אותה ומבצעים חיתוך של הקו עם הגבול של החלון ומחליפים את הנקודה שבחוץ בנקודה חדשה וחוזר חלילה (עד שמפינו בוודאות את כל הקווים). עבור פוליגונים: מסתכלים על כל גבול של החלון בנפרד. מתחילים מקודקוד כלשהו. בכל פעם שעוברים בין נקודה שנמצאת בתוך החלון לאחת שבחוץ, מוצאים נקודת חיתוך עם גבול החלון.

- **View Port Transformation** – מבצע מעבר מקואורדינאטות חלון לקואורדינאטות מסך.

- **Scan Conversion & Shading** – בחלק זה מחליטים איזה צבע כל פיקסל. יש 3 שיטות לחישוב Shading:

  - **Flat Shading** – בשיטה זו כל נקודה בתוך הפוליגון, מקבלת את נורמל הפוליגון.
  - **Gouraud Shading** – בשיטה זו מחשבים צבע לכל קודקוד (כולל תאורה) ובכל נקודה בפוליגון מבצעים ממוצע משוקלל של הקודקודים.
  - **Phong Shading** – עבור כל נקודה מוצעים נורמל המורכב מ"ממוצע משוקלל" של הנורמלים בקודקודי הפוליגון. לאחר קבלת הנורמל, מחשבים את הצבע של הנקודה.

Texture Mapping – כאשר מעוניינים להחליט על צבע של נקודה בפוליגון, משתמשים בצבע המתאים מהטקסטורה.

  - **Mapping** – ממפים את הטקסטורה למשטח ואז בשיטת uv בוחרים לכל נקודה במשטח את הצבע המתאים מהטקסטורה במהלך הScan Conversion. חיסרון של השיטה הוא בעת ביצוע טרנס' אפניות (כמו Shear) כתוצאה משגיאות שנוצרות באינטרפולציה הבי-ליניאריות. להשלים כיצד זה נפתר.

  - **Filtering** – לאחר קבלת המיקום שלנו בטקסטורה, יש להחליט באיזה צבע יצבע הפיקסל. פילטרינג מאפשר לנו לקבוע לאיזה חלק מהטקסטורה עלינו להתייחס (אם אנחנו קרובים למצלמה חלק קטן יותר ואם אנחנו רחוקים חלק גדול יותר). על החלק הרלוונטי מבצעים קונבולוציה כדי לקבל את צבע הפיקסל. כדי לבצע זאת ביעילות אנו נעזרים ב**MipMapping**. בשיטה זו המחשב שומר את הטקסטורה במספר גדלים מראש. בעת חיפוש צבע של נקודה נוכל לבחור את העתק עם הגודל הכי קרוב לגודל שאנו מחפשים ונדגום בהעתק זה את הפיקסל הכי קרוב אלינו (**Nearest Neighbor**) או נבצע אינטרפולציה בין 4 הפיקסלים הקרובים (**bilinear**). לחלופין, נוכל לבחור את 2 הגדלים התוחמים אותנו (ההעתק שגדול מאיתנו וההעתק שקטן מאיתנו) ולבצע ממוצע של הדגימות (**tri-linear**). בדר"כ משתמשים בשיטה האחרונה.
    שיטה נוספת שקיימת (ובדר"כ לא משתמשים) נקראת **Summed-area**. בשיטה זו מחשבים לכל פיקסל בטקסטורה (טקסל) את הסכום של הצבעים שמתחתיו ומימינו. עתה כאשר רוצים לדעת את הצבע של פיקסל שנופל בתוך מרובע בטקסטורה, ניתן במהירות למצוא את הסכום של הצבעים במרובע ולבצע ממוצע ע"פ מספר הפיקסלים שנמצאים במרובע זה בטקסטורה.

  - **Parameterization** – בשיטה זו אנו מבצעים מיפוי לפרימיטיב שלם ולא לפוליגון (למשל לצינור). ניתן לעשות זאת באמצעות הגדרת חתכים (בדר"כ במקומות נסתרים יותר) ועליהם להלביש את הטקסטורה. שיטה נוספת היא באמצעות אטלס: מחלקים את הצורה לחלקים ולכל חלק מגדירים טקסטורה. יתרון נוסף של האטלס הוא בכך שניתן לארוז את החתיכות ובכך לחסוך מקום.

אפליקציות לשימוש בטקסטורות

- באמצעות טקסטורה בשחור-לבן, ניתן ליצור משטחים בצבעים שונים. למשל, דוגמת עץ ניתן להלביש על משטחים בגוונים שונים של חום וע"י כך לקבל אפקט של עץ בצבעים שונים.

- ניתן להשתמש בטקסטורה כדי להשפיע על ערכים של החומר עצמו. למשל, ניתן להגדיר שהטקסטורה תקבע כמה המשטח ספקולרי וכו'.

- **Bump Mapping** – משתמשים בשיטה זו בטקסטורות כדי להשפיע על הנורמלים של המשולשים. ביצוע שיטה זו גורם לשינויי תאורה במשטח (למשל כדי לדמות קליפת תפוז).

- **Displacement Mapping** – האשליה דומה מעט ל-Bump Mapping. ההבדל הוא בכך שבשיטה זו מתבצע שילוש מחדש של ה-Mesh (כלומר כאן ה-Mesh עצמו הופך להיות "לא חלק" והשינוי אינו רק ברמת הנורמל).

- **Environment/Reflection Mapping** – מבצע מיפוי של דברים מהסביבה של העולם הוירטואלי (לוודא).

- **Image-Based Rendering** – משתמשים בצילומים כדי להלביש על מודל לא מדויק של העולם (כדי להוסיף למודל זה פרטים).

- **Solid Textures** - שימוש בטקסטורות תלת מימדיות כדי לקבוע צבע של כל פיקסל במשטח. דורש מקום רב לאחסון. ניתן לבצע גם באמצעות חישוב בזמן אמת.

Visible Surface Determination

לאחר החלטה איזה אובייקטים נמצאים בפיקסל מסוים, צריך להחליט איזה מהם להציג (מי הכי קרוב). כדי לפתור בעיה זו יש 2 גישות:

- **Image-precision algorithms** – בגישה זו אנו מחליטים עבור כל פיקסל איזה אובייקט מופיע בו.

- **Object-precision algorithms** – בגישה זו אנו מחלטים עבור כל אובייקט מה החלק הנראה שבו.

זיהוי משטחים הפונים אחורה
בגופים אטומים, לא ניתן לראות משטחים שפונים אחורה. כדי לזהות משטחים אלו משתמשים בסימן ה-Dot Product של נורמל המשטח עם וקטור הצפייה. בצורות קמורות, שיטה זו פותרת את כל הבעיה. באובייקט שאינו קמור, ייתכן ולא נראה חלק מהמשטחים שפונים קדימה.

- **Floating Horizon Algorithm** – בהינתן פונקציה ב-3 מימדים (x,y,z) אנו יוצרים מערך של ערכי x ו-z כך שבכל שורה ערכי z קבועים. בנוסף, אנו שומרים 2 מערכים: ערכי מקסימום ומינימום של Y. עתה אנו עוברים על כל השורות (החל מהשורה עם ערך z הקטן ביותר), בודקים האם הערך בשורה זו גדול מהמקסימום או קטן מהמינימום ומעדכנים את המערכים בהתאם.

- **Depth Sort** – בשיטה זו אנו לוקחים את כל הפוליגונים בסצנה ומציירים אותם לפי המרחק שלהם מהרחוק לקרוב (ע"י דריסה). יכולה להתעורר בעיה כי לא תמיד קיים סדר כזה. עם זאת, ניתן לזהות מצבים בעיתים ולחלק את הצורות הבעייתיות למספר צורות.

- **Z-buffer Method** – בשיטה זו סורקים את כל האובייקטים בסצנה באופן שרירותי. מוחזק באפר עם "המרחק" של כל פיקסל. עבור כל פיקסל של הצורה שאמור להיות מוצג, בודקים האם ערך Z של הפיקסל קטן מהערך בבאפר ורק עם כן, דורסים את הפיקסל השמור. מאוד קל למימוש ולכן בדר"כ ממשים בחומרה. לעומת זאת, חסרונות השיטה הם בכך שהיא דורשת הרבה מקום, הדיוק הסופי האפשרי בה יכול ליצור בעיות, יכול לגרום לבזבוז זמן בעת רנדור פוליגונים שבכלל לא מופיעים, דורש חישוב מחדש בעת שינוי גודל צורה וקשה לעשות שקיפות.

- **Scan Line Algorithm** – סורק את העולם שורה אחר שורה. לא משתמש ב-Z-באפר, כל שורה מחולקת למספר חלקים ובכל חלק מחלטים איזה פוליגון מוצג ולפי החלטה זו ממלאים את כל החלק בצבע. (לחזור לזה)

- **The BSP Tree** – מתחילים מפוליגון כלשהו ובונים עץ BSP עם פוליגונים שלפני הפוליגון ופוליגונים שמאחורי הפוליגון. פוליגונים שנמצאים באותו מישור איתו מחולקים ל-2. בסיום בניית העץ ניתן לטייל בו ולצייר את הפוליגונים מהרחוק לקרוב.

- **Area Subdivision Technique** – מתחילים מהמסך כולו. אם אין אף פוליגון במסך, צובעים בצבע הרקע. אם במסך יש רק פוליגון אחד המוכל, ממלא, או נחתך צובעים את החלק הרלוונטי. אם יש יותר מפוליגון אחד אך קיים פוליגון הממלא את כל המסך ויותר קרוב מכולם, צובעים את המסך בצבע שלו. אחרת, מחלקים את המסך ל-4 וחוזר חלילה.

OpenGL

הינו ממשק/סטנדרט/API המגדיר בערך 250 פקודות. המודל של OpenGL הוא פרוצדוראלי.
OpenGL היא מכונת מצבים. ה"מצב" בו נמצאת המכונה נשמר ב**glContext**. המצב כולל את הצבע הנוכחי בו מציירים, מטריצות מודל ופרוג'קטיון, שיטות ציור (פוליגונים, קווים וכו'), תאורה (מיקום ומצב), טקסטורות ועוד. המכונה מתחילה ממצב ברירת מחדל שהוגדר היטב. היתרונות בשימוש ב"מצב" הם כיוון ששיטה זו מאפשרת ניהול מספר חלונות, מדמה מצב של כמעט OOP, המשתמש לא צריך לזכור דבר וזה מאפשר אינטראקציה מהירה עם החומרה.
כדי לקבל ערכים של מצב מהמכונה ניתן להשתמש ב**glGet---**. זו קריאה יקרה.

לפי הקונבנציה של OpenGL, שם של כל שגרה מתחיל ב-gl, לאחריו מופיע שם הפונקציה ולבסוף מספר ואות המציינים את סוג הנתונים שמתקבלים. למשל glFunction2f היא פונקציה שמקבלת 2 משתנים מסוג float. הוספת האות v מסמנת שהמשתנים הם וקטורים (למשל glFunction2fv מקבלת 2 וקטורים של float). קבועים נכתבים באותיות גדולות עם התחילית GL_ (GL_CONSTANT).

Vertex – נקודה/קודקוד. רוב פעולות הציור מתבצעות באמצעותו. תכונות הפרימיטיב אותו אנו מציירים (קו/פוליגון/משולש) נקבעות ע"פ תכונות הנקודות. גודל מינימאלי של נקודה הוא פיקסל אך הגודל יכול להיות יותר גדול. מותר להשתמש ב-**glVertex** רק בין glBegin ו glEnd (לא עושה כלום אחרת).

פוליגון – מוגדר באמצעות נקודות וחייב להיות קמור, הנקודות חייבות להיות באותו מישור ואסור לגבולות 2 פוליגונים להיחתך. **glPolygonMode()** – איך לצייר את הפוליגון (קווים/מילוי/גם וגם).

GLU – ספריית העזרים של GL. מכיל, בין היתר, את GLU Quadrics שמשמש לצורך ציור צורות שונות (כמו כדור, צילנדר וכו').
JOGL – המימוש של OpenGL ב-Java.
OpenGLES – מימוש מצומם של OpenGL עבור מכשרים ניידים או אחרים שלא יכולים ליישם את הגרסה המלאה.
glGetError() – צריך לבדוק לפחות פעם בפריים כדי לוודא שלא היו שגיאות.

כיוון הפוליגונים – אם לא הוגדר אחרת, ציור הפוליגון נגד כיוון השעון יגרום לו להסתכל קדימה וציורו עם כיוון השעון יגרום לו להסתכל אחורה.

GL_CULL_FACE גורם לפוליגונים שמכוונים אחורנית לא להתרנדר.

טרנספורמציות - בכל זמן נתון יש מטריצת טרנספורמציה בזיכרון.
GL_MODELVIEW היא מטריצת הטרנס' שפועלות על האובייקטים בעולם.
GL_PROJECTION היא מטריצת הטרנס' שפועלת על המצלמה.

באמצעות פקודת **glMatrixMode()** אנו מחליטים על איזו מטריצה עובדים כעת.

glOrtho() – מגדיר את התצוגה להיות אורתוגראפית (כלומר הטלה מקבילית).
glFrustum(), glUPerspective(),glULookAt – מגדיר הטלה פרספקטיבית.

באפרים
מהווים חלק מהמצב של OpenGL. מימדי הבאפרים הם כמימדי שטח החלון עליו מציירים. קיימים מספר באפרים: צבע (Color), עומק (Depth), Accumulation Buffer, Stencil Buffer. **glClear** מנקה את הבאפרים לערך התחלתי.
לבאפר הצבע מגדירים באמצעות **glClearColor** מה הצבע אליו "מתנקה" הבאפר. חשוב לנקות באפר זה לפני כל ציור כי אחרת תתקבל מריחה.
Z-Buffer שומר את המידע על עומק כל פיקסל. מופעל ע"י **glEnable(GL_DEPTH_TEST).**
באמצעות **glClearDepth** מגדירים לאיזה ערך הבאפר "ינוקה" (כמעט תמיד 1).
glDepthFunc(GL_LEQUAL) מגדיר את ההשוואה בין הערכים (איזה פיקסל יבחר לתצוגה).

כאשר לא משתמשים בתאורה, צבע של כל נקודה הוא הצבע הנוכחי (לעומת לקיחת צבע מהחומר).
glShadeModel(GL_FLAT) – מגדיר שצבע של פוליגון יצבע ע"פ הנקודה האחרונה.
glShadeModel(GL_SMOOTH) – יבצע החלקה של צבעי כל הקודקודים המגדירים אותו.

האצת ציור
יש מספר שיטות להאצת הציור בOpenGL:

- **Display Lists** – יוצרים רשימה מוקלטת של פקודות והרשימה מועברת לכרטיס המסך. בכל פעם שקוראים לרשימה, היא מתבצעת אוטומטית בכרטיס המסך. הרשימה מקליטה פקודות OpenGL בלבד ולא פקודות Java. ישנן פקודות OpenGL שאינן מוקלטות ומבוצעות ישר.
- **Vertex Arrays** - כאשר רוצים לצייר אובייקט גדול (למשל שמורכב מהרבה משולשים), ניתן לבצע פקודה אחת של ציור ע"י העברת מערך עם מידע.

טקסטורות
יש בOpenGL אפשרות להשתמש בטקסטורות חד מימדיות, דו מימדיות ותלת מימדיות. JOGL מאפשר ליישם טקסטורות בפשטות. חשוב שתמונת טקסטורה תהיה בגודל שהוא חזקה של 2 (עבור תהליך ה-Mipmapping). לאחר עיבוד הטקסטורה, קואורדינאטות הטקסטורה הופכים להיות בין 0 ל-1.

GLSL
מאפשר לכתוב תכנות מחדש לכרטיס מסך כך שיעשה דברים שהוא בדרך כלל לא עושה (למשל, מימוש צללים). בעבר (ואפשר גם היום) היה צורך לכתוב ישירות לכל סוג של כרטיס מסך בנפרד. GLSL מאפשר לבצע את הכתיבה פעם אחת באמצעות API שכל כרטיסי המסך ממשים.

מידול
כיצד מייצגים, יוצרים ומשנים אובייקטים תלת מימדיים. יש 4 סוגים של ייצוג מודלים. בכל סוג ניתן לייצג כל דבר ולבצע עליו פעולות גיאומטריות. אנו זקוקים לייצוגים שונים כי לכל ייצוג יש את היתרונות שלו. ההבדלים בין האובייקטים נבדלים במהירות החישוב, הבדלים בגודל הזיכרון הדרוש, רמת דיוק, פשטות ושימושיות. סוגי הייצוג:

- **נקודות** – שיטה אחת הינה **Range Image**. בשיטה הזו סורקים אובייקט ויוצרים דגימות רבות של נקודות. לנקודות אלו מבצעים שילוש וכך נוצר המודל התלת מימדי. השיטה השנייה הינה **Point Cloud**. ???

- **משטחים**
  - בשיטת **Polygonal Mesh** מייצגים את המודל באמצעות צמתים וחיבור ביניהם. ניתן להשיג אותם ע"י ציור במחשב, סורקים שונים (לייזר, CAT וכו') או ע"י סימולציות.

- **Subdivision Surface** - בשיטה זו ניתן Mesh וחוקי חלוקה. הMesh וחלוקה זו יצרו מודל מוחלק סופי.
- **Parametric Surface** – בשיטה זו האובייקט מוגדר על ידי קבוצה של פונקציות רציפות המתחברות יחד.
- **Implicit Surface** – בשיטה זו מגדירים פונקציה. במקומות שהיא שווה ל-0 הגוף עובר. מקומות בהם היא קטנה מ-0 נמצאים בתוך הגוף ומקומות בהן היא גדולה מ-0 נמצאים מחוץ לגוף.

- **Solids**
  - **Voxels** – ייצוג אובייקט באמצעות קופסאות תלת מימדיות (בדומה לפיקסלים במישור דו מימדי).
  - **BSP Tree** – (תרי – מה זה?)
  - **CSG** – היררכיה של פעולות בוליאניות שמפועלות על צורות פשוטות היוצרות את המודל (למשל חיסור כדור מריבוע ייצור קופסא עם חור עגול בתוכה).
  - **Sweep** – (ליאור – מה זה?)

- **High-level structures**
  - **Scene Graph** (גרף סצנה) – שימוש באובייקטים כדי ליצור סצנה שלמה.
  - **Application Specific** – כל אפליקציה יכולה ליצור לעצמה שיטת ייצוג.

עיבוד Meshים

אנו רוצים לעבד Meshים כדי לתקן שגיאות, לצמצם את כמות החלקים (במקום המון משולשים מעט), הזזה של המודלים וכו'. רוב הפעולות על Meshים מתבצעות בשימוש במספר פעולות ברמה נמוכה של ה-Mesh: פיצול משטחים, פיצול Edge, הורדת Edge, איחוד קודקודים והסרת קודקודים.

ייצוג Meshים

ייצוג של ה-Mesh חייב לתמוך במספר פעולות נפוצות הדרושות לביצוע עם Mesh כמו ציור, הסרת קודקוד, החלקת אזור וכו'. בנוסף חשוב שה-Mesh יהיה יעיל מבחינת זיכרון. אפשרויות ייצוג:

- באמצעות **משטחים עצמאיים** – בכל משטח מציינים את הקודקודים שלו. החיסרון הוא שימוש חוזר באותם קודקודים וחוסר מידע על קשרים בין משטחים.
- **טבלת משחים וקודקודים** – חוסך מקום (של שיכפול הקודקודים) אך עדיין אין מידע על קשרים בין משטחים.
- **שימור כל הקודקודים, הקשתות והמשטחים השכנים** לכל <u>משטח</u> – יש מידע על קשרים בין משטחים אך זה דורש מקום רב נוסף.
- **שימור הקודקודים, הקשתות והמשטחים השכנים** לכל <u>קשת</u> – יותר יעיל מבחינת מקום.
- **Half Edge** - להשלים
- **לכל משטח לשמור את הקודקודים שלו והמשטחים השכנים**

Mesh segmentation

רוצים לחלק את הMesh לחלקים לפי קריטריון מסויים. הקריטריונים יכולים להיות: האם אנחנו רוצים שהגבולות יהיו חלקים? החלקים יהיו עגולים? חלקים גדולים/קטנים? מספר חלקים גדול/קטן? וכו' יש 2 סוגים לחלוקה:

- **Patch type** – חלוקה לפי המשטחים (לפי איך שזה נראה מבחוץ).
- **Part type** – חלוקה לפי נפח הצורה.

זו בעצם בעיית מינימיזציה. אנו רוצים לחלק את הצורה לחלקים תחת קריטריונים מסוימים. קיימת עבורנו פונקציה המחזירה ערך בהתאם לחלוקה ואנו רוצים למזער את ערך החלוקה תחת אילוצים שחייבים להתקיים. הבעיה הינה בעיה קשה ולכן יש לנו אלגוריתמי קירוב.

משתמשים בחלוקה זו כדי לנתח אובייקטים (זה הראש של האובייקט, זו הרגל וכו'), כדי לבצע דרמטיזציה קלה יותר (למשל הגדרת הטקסטורה של גב של 2 דינוזאורים מ-2 סוגים דומים), כדי להחיל תנועה על אובייקט מתנועה של אובייקט אחר, חיפוש אחר אובייקטים דומים וכו'.

Mesh Subdivision
כדי לבצע החלקה של ה-Mesh אנו משתמשים בחלוקה שלו. אנו לוקחים כל קשת ומוסיפים קודקוד לאמצע הקשת. לאחר מכן, כל קודקוד ב-Mesh (גם הקודמים) מעבירים לממוצע נקודות המרכז של כל המשטחים בהם הוא נוגע. בקצוות מבצעים טיפול מיוחד. יש בעיה בשיטה זו לשמר זוויות חדות.

Linear Subdivision – עובדת על Meshים עם מרובעים, כל קודקוד יכול לגעת משותף למספר בלתי מוגבל של מרובעים. כל מרובע מחלקים ל-4 (**Topology refinement**) ואז מבצעים ממוצע של כל קודקוד עם מרכזי כל המרובעים בהם הוא נוגע (**Geometry refinement**).

Catmull-Clark Subdivision – עובד על Meshים עם כל מיני צורות של פוליגונים. לאחר סיבוב אחד, כל הצורות משתנות למרובעים. מבצעים חלוקה של כל face לפי המרכז של הקשתות שמרכיבות אותו. לכל קודקוד שקיים ב-Mesh החדש נותנים ערך על פי הקודקודים המקיפים אותו בשיטה הבאה: עבור מרכזי משטחים נותנים משקל ממוצע של הקודקודים המקוריים של ה-Mesh. עבור מרכזי קשתות, מצבעים ממוצע משוקלל של שני הקודקודים על הקשת ושני מרכזי המשטחים. לנקודות המקוריות יש נוסחה מסובכת. ניתן בשיטה זו לבצע כמה שלבים בבת אחת באמצעות חישוב מתמטי. מובטח גזירות C^2 כמעט בכל מקום. בקודקודים עם דרגה שונה מ-4, מובטחת גזירות C^1. חלוקה זו מאוד נפוצה.

חלוקת פרפר – מחלקת Meshים המורכבים ממשולשים. נותנת חלקות C^2 כמעט בכל מקום ו-C^1 בקודקודים עם דרגה שונה מ-6.

Adaptive Subdivision – ברצוננו להחליק את ה-Mesh תחת הגבלה של מספר משולשים סופי. ניתן למדוד את מידת החלקות של החלוקה על פי מידת חלקות לוקלית ובאמצעות גודל השטח שתופס משולש "סופי". הפתרון: לעצור את החלוקה ברמות שונות במשטח. תנאי העצירה תלוי במידת החלקות. עם זאת, כתוצאה מרמות שנות של חלוקה עשויים להיווצר לנו "חורים" ב-Mesh. לכן, כדי לפתור זאת, מבצעים **חלוקה מאוזנת** (**Balanced Subdivision**). כלומר, אנו מבצעים חלוקה כך שיש הפרש של רמה אחת בלבד בין שני חלקים שונים ב-Mesh.

קיימות שיטות נוספות כמו **שיטת שורש 3** ושיטת **Vertex-split**.

Parametric Surfaces – שיטה להגדרת משטחים באמצעות פונקציות מדו-מימד לתלת מימד.

Cubic B-Spline – שיטה לקבל פונקציות עבור ה-Parametric Surfaces. בשיטה זו, אנו מקבלים שליטה מקומית, גזירות C^2, הקו המתקבל הוא מקורב ומתקיימת תכונת Convex Hull. כמו כן, הפולינומים הם מדרגה מדרג ה 3. כדי למצוא את הפונקציות, יש לנו נוסחה עם 16 דרגות חופש. האילוצים (החיבור של הנקודות חלק בפונקציה, בנגזרת הראשונה ובנגזרת הנישה) נותנים 15 דרישות. בדרגת החופש הנוספת אנו דורשים כי בנקודה 0, סכום הפונקציות יהיה 1.

Cubic Bezier – שיטה נוספת לקבלת הפונקציות עבור הParametric Surfaces. בשיטה זו אנו מקבלים שליטה מקומית, גזירות C^1, אינטרפולציה כל נקודה שלישית ומתקיימת תכונת Convex Hull. כמו כן, בנקודות הקצה האינטרפולציה היא מושלמת (כך ניתן לחבר אינטרפולציות ביזייר כדי לקבל קו שמקיים יותר אינטרפולציה). קיימת סימטריה בהגדרה.

Parametric Bicubic Patches – בדומה לSplines המקרבים קווים, שיטה זו נועדה לקרב Patch (חלק ממשטח תלת מימדי) באמצעות נקודות בקרה. השיטה משתמשת באחת השיטות לקירוב Splines. למשל, אם נשתמש במטריצה M במטריצת Bezier, נקבל Patch בו 4 הנקודות בפינה נמצאות בדיוק בנקודות הבקרה שלנו, הPatch יהיה Convex hull ויקיים שליטה מקומית. כדי ליצור רציפות בין Patchים, נבחר את אותם הנקודות בקצה שני Patchים מחוברים.

Feature	Polygonal Mesh	Parametric Surface	Subdivision Surface
Accurate	No	Yes	Yes
Concise	No	Yes	Yes
Intuitive specification	No	Yes	No
Local support	Yes	Yes	Yes
Affine invariant	Yes	Yes	Yes
Arbitrary topology	Yes	No	Yes
Guaranteed continuity	No	Yes	Yes
Natural parameterization	No	Yes	No
Efficient display	Yes	Yes	Yes
Efficient intersections	No	No	No

אנימציות וסימולציות

אנימציה היא שינוי לאורך זמן של אובייקטים לפי תסריט שנקבע מראש. **סימולציה** היא שינוי לאורך זמן של אובייקטים ע"פ כללים פיזיקאליים.

Keyframe Animation – קביעת מיקומים של אובייקט בנקודות זמן מסוימות (והמחשב מבצע אינטרפולציה בין המצבים הללו). אם לאובייקט מוגדר "שלד", ניתן להגדיר את הזוויות בין המפרקים בכל keyframe. המחשב יכול לחשב ע"י כך את המיקום של כל נקודה באובייקט. את השינוי בזווית לאורך זמן באמצעות Spline. אם נבצע אינטרפולציה לינארית בין הנקודות, לא נקבל מספיק חלקות בין ה-keyframes. עם זאת, אם נשתמש ב-Splines אנו עשויים ליצור "נקודות חדשות" שיגרמו לאובייקט להתנהג בצורה שלא התכוונו אליה (למשל שהמנורה תכנס לתוך הריצפה).

Adding inverse kinematics – בשיטה זו אנו מגדירים באיזה צורה אנו רוצים שהשלד יראה ואנו מעוניינים שהמחשב יחשב לבד את הזוויות (כיוון שלחשב זאת ידנית זה לא פרקטי). גם כאן את נקודות מיקום השלד ניתן לקרב באמצעות Splines. הבעיה היא שלפעמים קורה שהמיקום שלנו מכיל פחות נתונים ממספר המשתנים. במקרים אלו אומנם לא קיים פיתרון יחיד אך ניתן למצוא את הפיתרון הכי טוב.

Adding dynamics – אנו רוצים לדמות ריאליזם בתנועה (כלומר לקחת בחשבון חוקים פיזיקאליים). למחשב יש את התנועה המבוקשת ואת החוקים הפיזיקאליים והוא מוצא פיתרון אופטימאלי. כדי למצוא זאת, מתחילים מניחוש התחלתי עד שמגיעים לתנועה הטובה.