ВИСОКА ШКОЛА ЕЛЕКТРОТЕХНИКЕ И РАЧУНАРСТВА
СТРУКОВНИХ СТУДИЈА

Кузмић Немања

C++ апликација за генерисање сплајнова

- завршни рад -

Београд, октобар 2020.

Кандидат: **Кузмић Немања**

Број индекса: **НРТ-44/16**

Студијски програм: **Нове рачунарске технологије**

Тема: **C++ апликација за генерисање сплајнова**

Основни задаци:

1. Опис коришћене OpenGL библиотеке.

2. Имплементација C++ апликације за генерисање сплајнова.

3. Опис корисничког интерфејса.

Ментор:

Београд, октобар 2020 године.

_____

др Перица Штрбац, проф. ВИШЕР

РЕЗИМЕ:

У изради овог завршног рада користићемо C++ програмски језик у комбинацији са графичком библиотеком OpenGL и многим другим библиотекама као што су GLFW, GLEW и остале. Анализираћемо комплексност употребе OpenGL-а за исцртавање простих графичких примитива као што су троугао и квадрат. Да ли OpenGL и даље проналази употребу у 2020. години? Циљ овог рада је писање и тестирање функционалне апликације која ће нам помоћи да на основу задатих тачака и одговарајућег алгоритма израчуна потребне вредности и исцрта сплајн на екрану.

Кључне речи: C++, OpenGL, крива, сплајн, GLEW, GLFW

ABSTRACT:

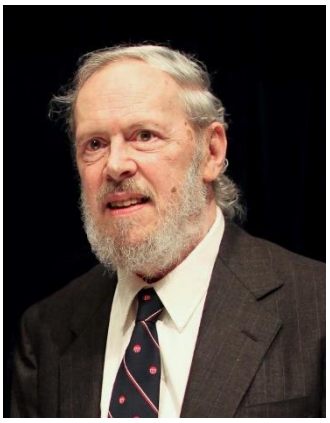
In making of this graduate work we will use C++ language together with graphics library OpenGL and libraries like GLFW, GLEW and others. We will analyze complexity of OpenGL usage in rendering graphical primitives like triangle and square. Is OpenGL still finding usage in 2020.? Assignment of this research to write and test functional application that will make us able to draw a spline according to given control points and suitable algorithms.

Keywords: C++, OpenGL, curve, spline, GLEW, GLFW

САДРЖАЈ:

1. ИСТОРИЈА И РАЗВОЈ ПРОГРАМСКОГ ЈЕЗИКА C++

1.1. ИСТОРИЈА C ПРОГРАМСКОГ ЈЕЗИКА



Слика 1.1.1. - Денис Ричи

Корени програмског језика C су везани за развој UNIX оперативног система, који је испрва написан у асемблерском језику на PDP-7 машини од стране Дениса Ричија (слика 1.1.1) и Кена Томпсона. Ускоро су одлучили да UNIX портују на PDP-11 машину. Прва верзија UNIX-а на PDP-11 је такође написана на асемблерском језику. [5]

Томпсон је желео флексибилнији програмски језик како би правио алате за нову платформу. Прво је покушао да користи Fortran компајлер, али је одустао од те идеје. Уместо тога, креирао је осиромашену верзију BCPL програмског језика. Карактеристике BCPL-а нису биле јавно доступне у то време. [6] Томпсон је модификовао синтаксу да буде мање речита и тиме је добио нешто слично B језику али једноставније. Свакако, мали проценат алата је написан на B-у зато што је био јако спор и није могао да експлоатише предности PDP-11 као што је адресабилност бајтова.

Ричи је 1972. Кренуо са радом на побољшању програмском језика B што је резултирало у новом језику - C. Компајлер за C и неколицина алата је укључена у “Version 2 Unix” оперативни систем. [7] Када је издат “Version 4 Unix” у новембру 1973. Године, Unix кернел је потпуно реимплементиран у C-у. Већ тада, C је имао веома јаке способности као што су struct типови.

1.2. ИСТОРИЈА C++ ПРОГРАМСКОГ ЈЕЗИКА

Слика 1.2.1. – Бјарн Строуструп

C++ је програмски језик вишег нивоа за опште намене. Креиран је од стране Бјарна Строуструпа (слика 1.2.1) као продужетак C програмског језика односно C са класама. У самом зачетку C++-а то је заиста била једина (битна) разлика између њих. Како је време одмицало, C и C++ су се дистанцирали и данас су то два различита језика са “истом” синтаксом али различитим стандардима.

C++ се развијао током времена и модерна верзија овог програмског језика је објектно-оријентисана, генеричка и функционална када је у питању манипулација “сирове” меморије. Увек је имплементиран као компајлирани језик и много компанија имају своја издања компајлера: Free Software Foundation, IBM, Microsoft, Intel и остали. Због тога је C++ доступан на великом броју платформи. [1]

Дизајниран је са благим заокретом ка системском програмирању и издвојеним (embedded) уређајима, за употребу са ограниченим ресурцима и великим системима. Као његове кључне особине се наводе перфомансе, ефикасност и флексибилност. [2]

C++ је нашао своју употребу и ван системског софтвера: користи се у десктоп апликација, видео играма и web серверима.

C++ је стандардизован од стране Међународне организације за стандардизацију (ISO) са најновијим стандардом ратификованим и издатим у децембру 2017. Године као ISO/IEC 14882:2017 у јавности познатији као C++17. Прва стандардизација догодила се 1998. Године као ISO/IEC 14882:1998 - познатији као C++98. Тренд је да се нови стандард издаје сваке 3 године. Рад на C++-у Строуструп је започео као део своје докторске дисертације 1979. Године.

Један од језика са којим је Строуструп имао прилике да ради је програмски језик звани Simula - чији назив имплицира да је био дизајниран за рад са симулацијама. Simula 67 је познат као један од првих програмских језика који је подржавао парадигму објектно-оријентисаног програмирања. Строуструп је открио да је ова парадигма јако корисна за развијање софтвера, али Simula је била превише спора за практичну употребу.

Започео је рад на "C са класама" и као што то и назив имплицира, замишљено је да то буде суперскуп C језика. Његова замисао је била да дода парадигму објектно-оријентисаног програмирања у C без компромитовања преносивости, брзине или способност за рад са "сировим" хардвером. Његов језик је укључивао класе, основно наслеђивање, inlining, подразумеване аргументе функције и строге провере типова варијабли.

Први компајлер за "C са класама" је био Cfront, који је наслеђен од C компајлера CPre. То је програм који је служио да преведе код C-а са класама у обичан C код. Јако занимљива је чињеница је да је и сам Cfront био написан углавном у C-у са класама чинећи га самоносећим компајлером (компајлер који може сам себе да компајлује). Cfront је напуштен 1983. Године након што је постало јако тешко интегрисати нове могућности у њега - првенствено C++ изузете (exceptions).

Име пројекта је промењено 1983. Године у "C++". Оператор ++ у C језику је оператор за инкрементирање променљиве, што указује на то како је Строуструп замислио свој пројекат. Наиме, уколико напишемо ++a, програм ће нашу променљиву а моментално инкрементовати и самим тим ми губимо оригиналност те промељиве. Када напишемо a++, наш програм ће прво сачувати ту промељиву, а затим је променити односно инкрементовати.

Након преименовања пројекта, много значајних могућности је додато. Најзначаније од њих су виртуелне функције, преоптерећење функција, референце са & симболом, const кључна реч и једнолинијски коментар са две косе црте - //. Ово означавање коментара је преузето из програмског језика BCPL.

Строуструп је 1985. Издао књигу "C++ Programming Language" која је служила као главна референца за употребу језика. Исте те године, C++ је представљен као комерцијални производ. Самим тим што језик још увек није био стандардизован, Строуструпова књига је била једина темељна и валидна документација. Језик је још једном надограђен 1989. Године са додатком protected и static чланова, као и вишестепено наслеђивање. [4]

1.3. ФИЛОЗОФИЈА

Кроз животни век C++-а, његов развој и еволуцију је одређивао сет принципа: [8]

- Мора бити заснован на реалним проблемима и све његове одлике би требало моментално бити корисне у већ постојећим програмима.
- Свака одлика би требало да има начин за имплементацију.
- Сваки програмер може слободно изабрати свој стил програмирања и тај стил би требало бити у потпуности подржан од стране језика.
- Убацивање корисне одлике је већег приоритета него спречавање сваке неправилне употребе C++-а.
- Требало би да достави механизме за организовање програма у засебно, јасно дефинисане делове и механизме за комбиновање засебно развијених делова.
- Без имплицитних повреда систеам типова (али дозволити експлицитну повреду - уколико је то експлицитно захтевано од стране програмера).
- Кориснички креирани типови морају имати исту подршку и перфомансе као и уграђени типови.
- Не би требало да постоји језик испод C++-а (осим асемблерског језика).
- C++ би требало да ради уз постојеће програмске језике, а не да форсира своје засебно и некомплатибилно програмерско окружење.
- Ако је намера програмера непозната, дозволити програмеру да је искаже предавањем мануелне контроле.

1.4. СТАНДАРДИЗАЦИЈА

C++ је стандардизован од стране акционе групе Интернационалне организације за стандардизацију (ISO), познатија као JTC1/SC22/WG21. За сада је издато пет ревизија C++ стандарда и у току је рад на следећој ревизији, C++20.

Прва стандардизација C++ се одвила 1998. Године као ISO/IEC 14882:1998. Након пет година, издата је нова верзија стандарда названа ISO/IEC 14882:2003 која је поправила проблеме који су настали употребом C++98 верзије стандарда.

Следећа ревизија стандарда је неформално названа "C++0x" јер је било очекивано да ће бити издата до краја 2009. Године али се то тек десило 2011. Године. C++11 (14882:2011) је укључивао доста адиција и језгру језика и стандардној библиотеци.

Као део процеса стандардизације, ИОС такође издаје и техничке извештаје и спецификације:

- **ISO/IEC TR 18015:2006** - Употреба C++ на издвојеним системима и импликације перфоманси C++ језика и могућности библиотеке.
- **ISO/IEC TR 19768:2007** - Познатији и као "C++ технички извештај 1" о екстензијама библиотеке које су углавном интегрисане у C++11.
- **ISO/IEC TR 29124:2010** - Специјалне математичке функције.
- **ISO/IEC TR 24733:2011** - Аритметика децималне плутајуће тачке.
- **ISO/IEC TS 18822:2015** - Стандардна библиотека за рад са системом датотека.
- **ISO/IEC TS 19570:2015** - Паралелне верзије алгоритама стандардне библиотеке.
- **ISO/IEC TS 19841:2015** - Софтверска трансакционална меморија.
- **ISO/IEC TS 19568:2015** - Нови сет екстензија за библиотеку, од којих су неки већ интегрисани у C++17.
- **ISO/IEC TS 19217:2015** - C++ концепти, који су интегрисани у C++20.
- **ISO/IEC TS 19571:2016** - Библиотечне екстензије за конкурентни рад.
- **ISO/IEC TS 19568:2017** - Нови сет екстензија за библиотеку опште намене.
- **ISO/IEC TS 21425:2017** - Библиотечне екстензије за распоне - интегрисано у C++20.
- **ISO/IEC TS 22277:2017** - Корутине.
- **ISO/IEC TS 19216:2018** - Библиотека за умрежавање.
- **ISO/IEC TS 21544:2018** – Модули [9]

1.5. ЗАЈЕДНИЦА

1.5.1. C++ Core Guidelines

Бјарне Строуструп и Херб Сатер су покренули колаборативну иницијативу звану "C++ Core Guidelines". То је резултат много година дискутије и дизајна у великом броју организација. Њихов дизајн охрабрује општу применљивост и широку адаптацију али може бити слободно измењен за потребе одређене организације.

Циљ иницијативе је да помогне људима да користе модерни C++ ефикасно. Модерним C++-ом се сматра C++11, C++14 и ускоро C++17. Смернице су углавном намењене за проблеме вишег нивоа, као што су интерфејси, управљање ресурсима, управљање меморија и конкуренцију. Таква правила имају утицај на архитектуру апликације и дизајн библиотеке. Поштовање смерница довешће до кода који је сигуран за статичке типове, нема цурење ресурса и обезбеђује механизме који ће довести до бољег примећивања логичких програмским грешака које су уобичајене у коду данас. [10]

1.5.2. Критичка мишљења

"*C++ је уистину постао превише наклоњен ка експертима - поготово у време када је степен ефективне формалне едукације просечног софтвер инжењера посустао. Свакако, решење није да се програмски језици углупе већ да се користе различити програмски језици и да се обучи више експерата. Мора да постоје језици за експерте - и C++ је један од тих језика.*"

"*Моја жеља је била да C++ првенствено дизајнирам као језик за системско програмирање: желео сам да будем у могућности да пишем драјвере, софтвер за издвојене системе и остали код којем је потребно да користи хардвер директно. Следеће, желео сам да C++ буде добар језик за дизајнирање алата. То је захтевало флексибилност и перфомансе, али и могућност да се изразе елегантни интерфејси. Мој поглед је био да би Ви радили ствари вишег нивоа, да напишете комплетне апликације, Ви бисте прво требали да купите, напишете или позајмите библиотеке које би Вас снадбеле одговарајућим апстракцијама. Често, када људи имају проблема са C++-ом, прави проблем је да они немају одговарајуће библиотеке - или не могу пронаћи библиотеке које су доступне.*"

"*Остали језици су покушали да директније подрже апликације високог нивоа. То функционише, али често та подршка носи цену специјализације. Лично, ја не бих дизајнирао алат који би могао да уради само оно што ја желим - циљам на уопштеност.*" [11]

"*Мислим да је [прављење програмских језика лакшим за просечне људе] заблуда. Идеја програмирања као полупрофесионалним задатаком, примењена од стране људи са тренингом од неколико месеци, је опасна. Не бисмо толерисали водоинсталатере или рачуновође са лошом едукацијом. Немамо за циљ да архитектура (зграда) или инжењеринг (мостова и возова) постану приступачнији људима са прогресивно мање тренинга. Уистину, један озбиљан проблем је да је тренутно, превише подедукуваних и недовољно утрениpаних софтвер инжењера.*" - Бјарне Строупстроу [12]

2. ОСНОВЕ РАЧУНАРСКЕ ГРАФИКЕ

Рачунарска графика је уметност исцртавања слика на рачунарским екранима уз помоћ програмирања. Укључује израчунавања, креирање и манипулацију података. Можемо рећи да је рачунарска графика алат за рендеровање уз помоћ којег генеришемо и манипулишемо сликама. [13]

Рачунарска односно дигитлна слика је сачињена од одређеног броја пиксела. Пиксел је најмања адресабилна графичка јединица репрезентована на рачунарском екрану. [14] Данас, рачунарска графика је једна од кључних технологија у дигиталној фотографији, филму, видео играма, мобилним телефонима и рачунарским екранима као и у многим специјализованим апликацијама, као на пример за симулацију. Велика количина специјализованог хардвера и софтвера је развијена и екране већине данашњих уређаја контролише графички хардвер. Израз рачунарска графика је први пут искоришћен 1960. Године од стране истраживача Верна Хадсона и Виљема Фетера из Боинга.

Рачунарска графика је одговорна за приказивање уметности и података слика ефикасно и значајно - или као тачка комуникације између рачунара и корисника. Такође се користи за анализу и обраде података слика добијених из физичког света. Развој ове технологије је довео до значаној утицаја на много типова медија и изазвао је револуцију у сферама анимације, филмова, реклама, видео играма и графичком дизајну уопштено.

Термин рачунарска графика означава неколико различитих ствари:

- Репрезентација и манипулација података слика од стране рачунара
- Разне технологије помоћу којих се креирају и манипулише сликама
- Област рачунарске науке која проучава методе за дигиталну синтезу и манипулацију визуелних садржајем [15]

Много алата је развијено за визуелизацију података. Рачунарски генерисане слике могу бити подељене у неколико различитих типова: дводимензионални (2D), тродимензионални (3D) и анимирана графика. Како је технологија напредовала, 3D рачунарска графика је постала уобичајена али 2D рачунарска графика је и даље у широкој употреби.

У протеклој деценији, неколико нових поља за специјализацију је настало као што су: визуелизација информација и научне визуелизације које се више баве са "*визуелизацијом тродимензионалног феномена* (архитектонски, метеролошки, медицински, биолошки итд.) *где је нагласак на реалном рендеровању запремина, површи, изворе светлости уз можда динамички компоненту* (времена)". [16]

2.1. ИСТОРИЈА РАЧУНАРСКЕ ГРАФИКЕ

Напретци у проучавању електротехнике, електронике и телевизије током прве половине 20. века су били зачетци поља проучавања рачунарске графике. Прва катодна цев, Браун цев, је осмишљена 1987. И дозволила је графички адаптацију осцилоскопа и војних контролних табли. [17] Војна индустрија је имала значајнији и директнији утицај у настанку рачунарске графике. Развијени су први дводимензионални електронски дисплеји који су динамички одговарали на програмски или кориснички унос.

Ипак, рачунарска графика остаје затворена наука до периода након Другог светског рата - током тог времена наука се формирала из комбинације универзитетског и лабораторијског академског истраживања ради напреднијих рачунара. Други део слагалице представља жеља војске Сједињених Америчких Држава ка даљем усавршавању кључних технологија попут радара, напредне авијације и напреднијих ракета. Нови типови дисплеја су били неопходни како би се обрадиле богати и позамашни подаци добијени радом на тим пројектима што је проузроковало коначно формирање рачунарске графике као науке.

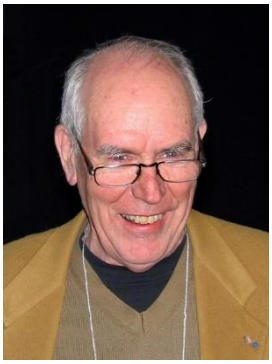
2.1.1. Светлосна оловка

Рани пројектни као што су Whirldwind и SAGE Projects су представили катодну цев као одржив дисплеј и интерфејс за интеракцију. Убрзо су и представили светлосну оловку (слика 2.1.1.1) као уређај за унос информација. Једна од првих видео игара која је представљала препознатљиву, интерактивну графика је била “Тенис за двоје” - креирана је за осцилоскоп од стране Виљема Хигинботама како би забавио посетиоце у Брукхавен Националној лабораторији 1958. Године. То је била симулација тениског меча. Даглас Т. Рос је 1959. дао значајан допринос тако што је током рада на МИТ-у трансформисао математичке изразе у компјутерски генерисане 3D векторе и ту прилику је искористио како би креирао слику карактера из Дизни цртаћа. [18]

Нова компанија у електроници Хевлет-Пакард је озваничена 1957. године након што се развијала претходну деценију и остварила је значајне везе са Станфорд Универзитетом кроз њене основаче - иако су похађали наставу али нису дипломирали. Ово је започело деценијама дугу трансформацију залива Сан Франциска у светски познату област за развој рачунарских технологија - сада познатија као Силиконска Долина. Поље рачунарске графике се развијало раме уз раме уз развој хардвера за рачунарску графику.

Слика 2.1.1.1. - Светлосна оловка

2.1.2. Иван Сатерленд

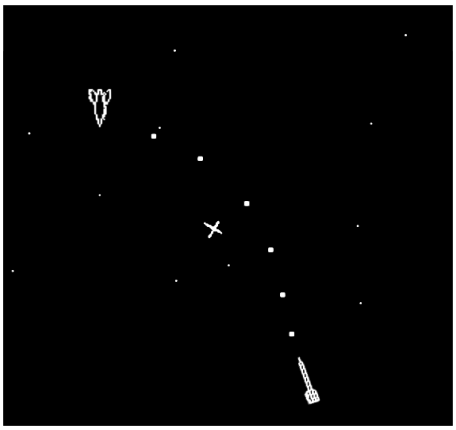


Слика 2.1.2.1. - Иван Сатерленд

Даљи напретци у рачунарству су довели до значајнијих достигнућа у интерактивној рачунарског графика. TX-2 рачунар је развијен 1959. Године у МИТ-овој Линколн лабораторији. TX-2 је представио број нових човек-машина интерфејса. Светлосна оловка је могла бити коришћена за цртање по рачунару користећи револуционарни Скечпед софтвер Ивана Сатерленда (слика 2.1.2.1).

Уз коришћење светлосне оловке, Скечпед је дозволио кориснику да црта једноставне облика на рачунарском екрану, да их сачува и позове касније. Светлосна оловка је имала малу фотоелектричну ћелију у свом врху. Ова ћелија је емитовала електронски импулс сваки пут када је постављена испред рачунарског екрана и проузрокује фокусирање и моментално пуцање електронског топа. Једноставним временским мерењем електричног импулса са тренутном локацијом електронског топа, постало је лако одредити где се оловка налази на екрану у сваком моменту. Када је то одређено, рачунар би исцртао показивач на тој локацији. Сатерланд је једноставно проналазио савршена решења за сваки графички проблем са којим се срео. Чак и данас, многи стандардни интерфејси рачунарске графике могу повући паралелу са овим историјским Скечпед програмом. Један од примера је исцртавање ограничења односно графника. Ако програмер жеи ли да исцрта квадрат, он не мора да се бави повлачењем четири савршене линије. Једноставно може исказати жељу за цртањем квадрата и у програм унети жељену локацију и димензију квадрата. Конструкцију савршене слике би преузео и извршио програм. Још један пример су Сатерландови софтверски моделирани објекти - не слика објеката. Уколико би неко желео да повећа прозоре на цртежу своје слике - једноставно ће увећати само прозоре без деформације остатка фасаде. [19]

2.1.3. Разрада рачунарске графике



Слика 2.1.3.1. - Спејсвор

Још један студент МИТ-а Стив Расел је 1961. креирао видео игру која ће у историји остати упамћена као прва видео игра, иако је Тенис за двоје игра била старија три године. Игра Спејсвор! (слика 2.1.3.1) је била написана за ДЕК-ов ПДП-1 рачунар и постала је моментални успех. Након широке дистрибуције игре међу корисницима ПДП-1, ДЕК је откупио права на њу. Инжењери су је користили као дијагностички програм на сваком новом примерку ПДП-1. Продајни тим није пропустио прилику да својим купцима понуди и "прву видео игру на свету".

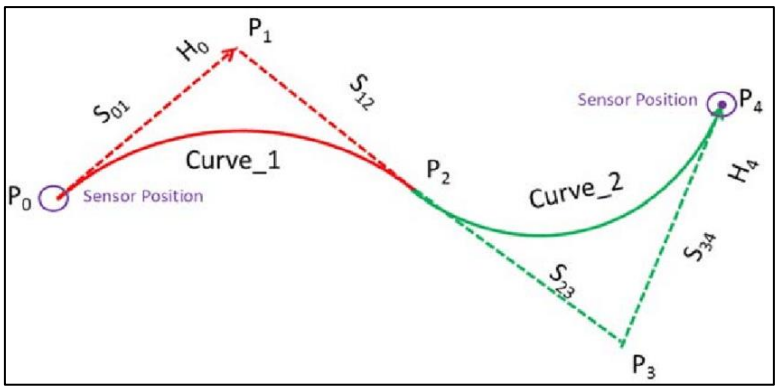
Видео игру Спејсвар! је и данас могуће играти скоро у свом оригиналног облику на следећем линку:

https://www.masswerk.at/spacewar/

У исто време, на Кембриџ универзитету, Елизабет Валдрам је написала код за приказивање радио-астрономских мапа на катодној цеви. [20]

У Бел телефонској лабораторији, три научника, Кен Кноултон и Мајкл Нол су започели рад у сфери рачунарске графике. Синден је креирао филм зван "Сила, маса и кретање" илуструјући Њутнове законе у пракси. Доста научника је у то време користило рачунарску графику како би представили своја истраживања. У Беркли лабораторији, Нелсон Макс је креирао филмове "Ток вискозних течности" и "Пропагација ударних таласа у чврстој форми". Боинг је креирао филм "Вибрације ваздухоплова".

Раних 1960-тих, рачунарска графика је такође имала утицај и у аутоиндустрији захваљујући раду Пјера Безиера у Реноу. Он је користио криве Пола де Касељауа - сада се оне зову Безиерове криве (слика 2.1.3.2) захваљујући његовом раду у струци како би развио тродимензионалне технике моделирања за ауто шасије. Безиерове криве представљају темељ за криво-моделијурући рад у струци, како су криве, за разлику од полигона, математички сложени ентитети за цртање и моделирање.

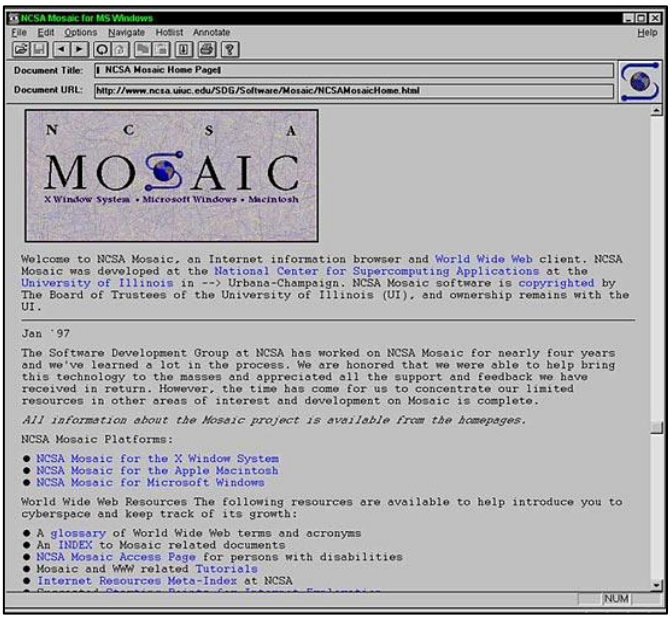


Слика 2.1.3.2. - Безиерова крива

2.1.4. Временска линија развоја рачунарске графике

- 1951. године - Џеј Форестер и Роберт Еверет са МИТ-а осмишљавају Вирлвинд, мејнфрејм рачунар који је у могућности да прикаже примитивне слика на телевизијском монитору или на визуелној приказној јединици.
- 1955. године - Директан наследник Вирлвинда, МИТ-ов ПЗО (Полуаутоматска земљана опрема) рачунар користи једноставну векторску графику како би приказао слике радара и постаје кључан део противваздушне одбране САД-а.
- 1959. године - Џенерал Моторс и ИБМ развијају Дизајн потпомогнут рачунаром (ДПР) систем како би олакшали инжењерима дизајнирање аутомобила.
- 1961. године - Студент МИТ-а Стив Расел је развио Спејсвар! Видео игру на ДЕК-овом ПДП-1 минирачунару.

- 1963. године - Иван Сатерленд, зачетник поља интеракција човек-рачунар (олакшавајући употребу рачунара људима) развија Скечпед, један од првих рачунарских-потпомогнутих пакета за дизајн у ком су слике могле бити исписане помоћу светлосне оловке.
- 1965. године - Хауард Вајз одржава егзибицију уметничких дела креираних уз помоћ рачунара у његовој галерији у Њујорку.
- 1966. године - НАСА развија програм за обраду слике назван ВСКП (Видео слика комуникација и пријем), који се покретао на ИВМ мејнфрејмовима са наменом да обрађује слике месеца које су прикупљале свемирске летелице.
- 1970. године - Безиерове криве су развијене, веома убрзо постају незаобилазни алат у векторског графици.
- 1980-тих година - Појава повољног, лаког за коришћење Епл Мекинтош рачунара отвара пут за десктоп паблишинг (самостално развијање ствари на вашем малом кучном рачунару).
- 1985. године - Мајкрософт развија прву верзију једноставног растерског програма за цртање имена МС Пејинт.
- 1990. године - Адоби развија прву верзију Фотошопа, данас најпознатијег пакета за професионалну обраду слика.
- 1993. године - Студент Марк Андерсен развија Мозаик (слика 2.1.4.1), први веб браузер који је био у могућности да прикаже и текст и слике једно до другог што је довело до огромног пораста интересовања за веб. [21]



Слика 2.1.4.1. - Мозаик веб претраживач

2.2. КОНЦЕПТИ РАЧУНАРСКЕ ГРАФИКЕ

2.2.1. Врсте слика

2.2.1.1. Дводимензионална слика

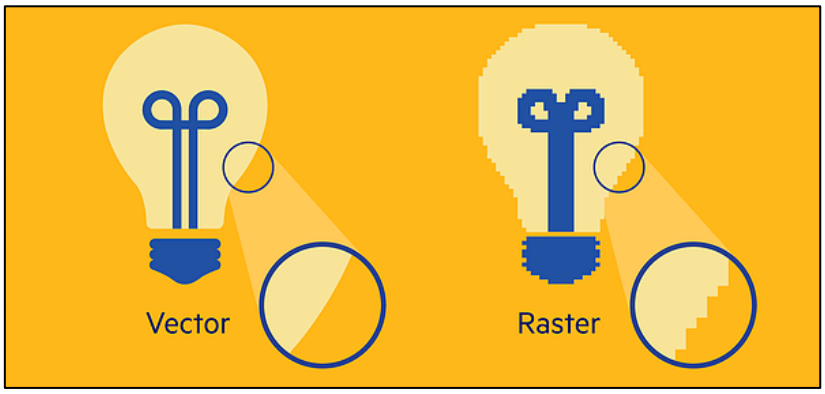
Дводимензионална рачунарска графика је генерација дигиталних слика заснованих на коришћењу рачунара. Најчешће се користи у апликацијама које су развијене како би пресликале технологије штампања и цртања као што је типографија. У овим апликацијама, дводимезионална графика не представља само објекат из реалног света већ и независни артифакт са придодатом семантичком вредношћу. Дводимензионални модели самим тим пружају директнији контролу над својствима графике него тродимензионала графика – чији је приступ погоднији фотографији.

Пиксел арт је вид дигиталне уметности која се креира помоћу софтвера за растерску графику, где се графике измењују на нивоу пиксела. Графика у већини старих рачунара и видео игара је пиксел арт.

Спрајт графика је дводимензионлна графика или анимација која је интегрисана у већој сцени. Иницијално је подразумевало само графичке објекте који су обрађени независно од меморијске битмапе видео дисплеја. Сада подразумева разне начине графичких слојева. Првенствено, спрајтови су били метода интегрисања неповезаних битмапа како би изгледали као да су део нормалне битмапе на екрану, као што је креирање анимираног играча којег је могуће померати по сцени без утицаја на податке који дефинишу већину сцене.

Растерска графика је тачкаста матрица која углавном представља правоугаону мрежу пиксела која је у целини видљива на неком медијуму за приказивање. Оне су складиштене у разним форматима (.jpeg, .png). Растер је углавном дефинисан ширином и дужином графике у пикселима као и бројем битова по пикселу илити дубина боје – представља колико разноврсне боје може презентовати. [22]

Формати векторске графике су комплементарни растерској графици (слика 2.2.1.1.1). Векторска графика се састоји у кодовању информација о облицима и бојама које сачињавају графику, што доводи до веће флексибилности у рендерингу. Постоје случајеви када је најбоље радити са растерском графиком а постоје и обрнути случајеви – зависи од ситуације. [23]

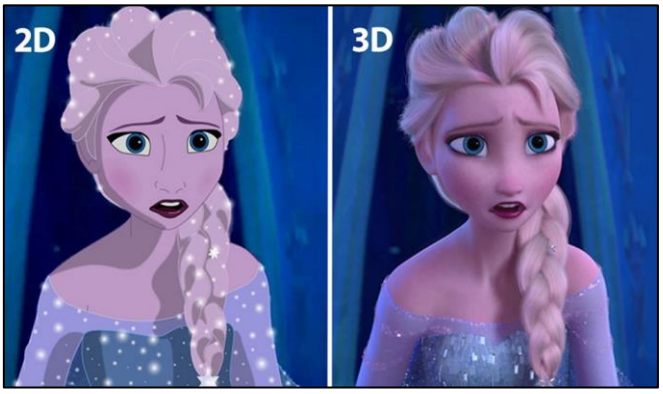


Слика 2.2.1.1.1. – Разлика између векторске и растерске графике

2.2.1.2. Тродимензионална слика

Тродимензионалне графике користи тродимензионалну репрезентацију геометријских података. Због перфомански ово се складишти на рачунару. Упркос овим разликама, тродимензионална рачунарска графика се ослања на сличне алгоритме које користи и дводимензионална графика у коначном изрендерованом фрејму.

У графичком софтверу, разлика између ова два типа графике се понекад утопи (слика 2.2.1.2.1) – дводимензионалне апликације могу користити тродимензионалне технике како би постигли ефекте као што је осветљење. Тродимензионални модел може исто што и тродимензионална графика са тим што модел може бити ускладиштен на датотеци – он постаје графика након рендеровања (исцртавања на екрану).

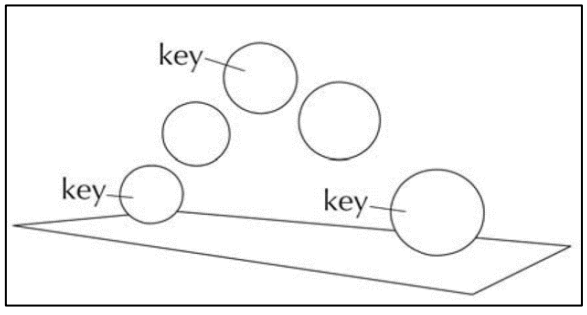


Слика 2.2.1.2.1. – Разлика између дводимензионалне и тродимензионалне слике

2.2.1.3. Рачунарска анимација

Рачунарска анимација је уметност креирања слика у покрету помоћу рачунара. Углавном се то односи на тродимензионалне слике али и даље се анимирају дводимензионални елементи поготово када су перфомансе у питању. Виртуелни ентитети могу садржати и могу бити контролисани уз помоћ намењених атрибута као што су вредности трансформације (локација, оријентација, скалирање) које су сачуване у објектовој трансформационој матрици. Анимација је промена атрибута у времену.

Основни начин креирања анимације је уз помоћ кључних фрејмова (слика 2.2.1.3.1), од којих сваки садржи различите вредности у одређеном тренутку времена – софтер за анимацију интерполира све вредности атрибута између два кључна фрејма. Да би креирали илузију кретања, слика је приказана затим је брзо замењена са новом сликом која је благо измењена што људско око не може приметити (уколико је фрејмрејт изнад 23).



Слика 2.2.1.3.1. - Кључни фрејмови у анимацији

2.2.2. Градивни елементи

У рачунаркој графици, основни градивни елемент сваке растерске слике је пиксел (pixel - picture element). Пиксели су смештени на дводимензионалну мрежу и углавном су репрезентовани користећи тачке или квадратиће. Сваки пиксел је узорак изворне слике са тим што више узорака може пружити прецизнију репрезентацију изворне слике. Интезитет сваког пиксела је варијабилан - у мултиколор системима сваки пиксел описују три компоненте као на пример црвена, зелена и плава (RGB систем) мада постоји више различитих модела боја (CMYK, HSV). [24]

Примитиве су просте јединице које графички систем може комбиновати како би креирао сложеније слике или моделе. Полигони или троуглови (углавном троуглови) у тродимензионалном рендерингу представљају примитиве. Примитиве могу бити подржане од стране хардвера за ефикасно рендеровање или за градивне блокове у графичкој апликацију.

Тачке су јединствено дефинисане од стране њихових X и Y координата. Углавном се не исцртавају самостално већ се користе у функцији описа других објеката као што су линије које су дефинисане њиховим почетним и крајњим тачкама.

Линије, полилиније или криве могу бити дефинисане помоћу две или више тачака. Линије захтевају две тачке. Криве захтевају две тачке и додатне контролне тачке. Полилиније су повезане секвенце линија.

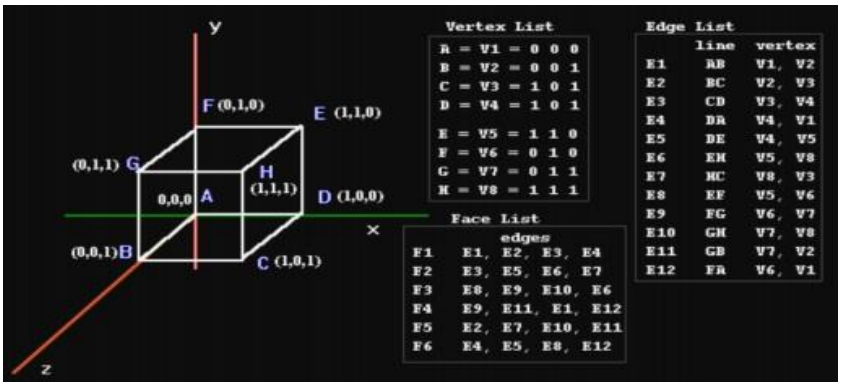
Подручја су ограничена затвореним полилинијама или полигонима. Могу бити попуњене са бојом или текстуром.

Линије, поготово праве линије сачињавају основан градиван блок рачунарских слика. Пример су линијски графови, пита и бар графикони, дводимензионали и тродимензионални графови математичких функција, инжењерских цртежа и архитектонских нацрта, Права линија је толико основна у креирању слика да спада у графичке примитиве.

Може бити развијена на два начина:

- Структурална метода одређују који пиксели морају бити сетовани пре исцртавања линије
- Условна метода тестира одређене услове да би пронашла који је следећи пиксел за сетовање

Полигон, иако се конструише од правих линија, је јако важна графичка примитива. Често желимо да се опходимо према полигону као према недељивом ентитету пошто се слике објеката из реалног света састоје од великог броја полигона. Полигон је затворено подручје слике ограничено правим линијама или кривама, испуњено једном бојом. Како су слике дводимензионалне, полигон је затворена планарна фигура. Самим тим, имплементирање полигона као графичке примитиве је природно и корисно. Можемо дефинисати полигон као слику која се састоји од коначног уређеног скупа правих граница званих ивице. Исто тако, полигон може бити дефинисан сортираног секвенцом темена, крајева полигона (слика 2.2.2.1). Ивице полигона су онда прикупљене пролазом кроз темена у задатом реду.



Слика 2.2.2.1. - Листа темена, наличја и ивица полигона

Листа ивица је сасвим довољна за вајрфрејм исцртавања (жичано-мрежасти приказ). Два узастопна темена дефинишу једну ивицу. Полигон затварамо тако што повежемо последње и прво теме. Листа наличја је потребна за попуњавање полигоне. Можемо декомпозирати сцену из реалног света у колекцију полигона једноставних облика. Пример – једноставна кућа може бити конструисана уз помоћ квадрата и правоугаоника. Свакако, нити праве линије нити полигони могу прецизно описати сцену из реалног света – то је само апроксимација сцене – реалне сцене су фракталне природе – за сада их је немогуће пренети у дигитални облик. [25]

Рендеринг или исцртавање представља генерисање дводимензионалне слике од тродимензионалних модела у домену рачунарских програма.

Датотека сцене садржи објекте у прецизно одређеном језику или структури података; геометрију, тачку гледишта, текстуре, осветљење и сенчење. Ови објекти представљају опис циртуалне сцене. Подаци су прослеђени програму за рендеровање за обраду и излаз у виду дигиталне слике или растерске датотечне слике. Рендерер је обично уграђен у програм за рад са рачунарском графиком (у развоју видео игара то би био енџин - погон, агрегат, мотор). Термин рендеринг може бити у аналогији са “уметниковим исцртавањем” - то представља цртање нечега што је уметник видео или тренутно гледа. Иако технички детаљи рендеринга могу варирати од технологије до технологије, кораци од почетка до краја процеса су уједињени у термину графички пајплајн (графичка цев) који води до хардверског уређаја за рендеровање као што је графичка картица. Графичка картица је уређај специјализован за графичка израчунавања - његова примарна сврха је асистенција централној процесорској јединици у калкулацијама.

Рејтрејсинг или праћење зрака представља технику из породице алгоритама за детерминацију распореда слике за генерисање слике путем праћења зрака светла кроз пикселе у сликовној равни. Ова техника је способна да произведе високе нивое фотореалистичости али истискује високу цену компутационе снаге.

Сенчење означава детерминацју дубине светлости тродимезионалних модела или илустрација - одређивање нивоа мрачности. То је процес који се користи у цртању тј. приказивању сенки на папиру тако што се слика појачава у сегменту који се налази у мрачнијем пределу - исто тако предео који се налази на светлом месту ће бити нешто слабије обојен представљајући већу количину светлости у том сегменту.

Елементи који су заслужни за израчунавање сенки у тродимезионалном програмима се зову шејдери за сенку. Квар шејдера за сенку се може приметити уколико је цела сцена у константом тону светла (слика 2.2.2.2).

Слика 2.2.2.2. - Недостатак шејдера за сенку на левој слици

Мапирање текстура је метода за додавање детаља, текстуре површи или боје на компјутерску генерисану слику или тродимензионални модел. Зачетник мапирања текстура је др. Едвин Катмул. Текстурна мапа је примењена на површину облика или полигона. Овај процес је сличан лепљењу папира на ком су исцртани обрасци на обичну белу кутију. Процедуралне текстуре су креиране изменом алгоритамских параметара који се користе за генерисање излазне текстуре. Битмап текстуре су креиране у рачунарском програму или увезене из неког дигиталног медијума за фотографисање. Ове две методе су највише коришћене за имплементирање дефинције текстуре на тродимезионалне моделе. За најпрецизнију и најреалистичнију примену текстура на површ модела су потребне технике као што је УВ мапирање (ручни распоред текстурних координата) за полигонске површи, док неуниформне рационалне Б-сплајн (NURB) површи имају своју интристичку параметаризацију која се користи за текстурне кооридинте. Мапирање текстура као дисциплина садржи још неке технике за што реалистичнији резултат рендеринга: креирање нормалних мапа и бамп мапа које се примењују на текстуру како би симулирали висину, спекуларне мапе за симулацију сијања и одсјаја.

Рендеровање ентитета који су независни од резолуције за приказ на растерском уређају као што је екран са течним кристалима неизбежно проузрокује назубљење по геометријским ивицама и по ивицама текстурних детаља - ови артифакти се називају џегији. Разни алгортимни могу бити примењени - као што је суперсемпловање и након тога прилагођени по жељи корисника како би се добило најефикасније и најпријатније решење.

3. ГРАФИЧКА БИБЛИОТЕКА OPENGL

3.1. УВОД У OPENGL

OpenGL (Open Graphics Library) се посматра као апликациони програмски интерфејс (API). Постоји више нивоа апстракције за тај интерфејс али у самом корену се користи C API. OpenGL API је state machine - машина са коначним бројем стања. Свака од функција сетује или враћа неко стање у OpenGL. Једине функције које не утичу на тренутно стање су фукнције које користе кренутно стање како би покренуле процес рендеринга.

Машина са коначним бројем стања се може описати као јако велика структура са јаком великим бројем различитих поља. Ова структура се зове OpenGL контекст и свако поље у контексту репрезентује неку информацију која је неопходна за рендеринг. API у C је дефинисан бројем дефиниција типова, #define вредностима енумератора и функцијама. Дефиниције типова уводе основне GL типове као што су GLint, GLfloat и остале. Оне су дефинисане да имају одређену битску дубину. Комплексни агрегати као структуре никад нису директно изложене, такве конструкције су мудро скривене иза апликационог програмског интерфејса. Ово омогућава лакше експонирање OpenGL позива према другим програмским језицима без комплексног слоја конверзије.

У C++ уколико желимо да креирамо објекат који садржи цели број, реални број и текст то бисмо урадили на начин приказан на слици 3.1.1.

```
1   struct Objekat {
2       int broj;
3       float prozirnost;
4       char* ime;
5   };
6
7   // Kreiramo skladište za objekat.
8   Objekat noviObjekat;
9
10  // Upisujemo podatke u objekat.
11      noviObjekat.broj = 10;
12      noviObjekat.prozirnost = 0.66f;
13      noviObjekat.ime = "Neki tekst."
```

Слика 3.1.1. - Креирање једног објекта у C++

Међутим, у OpenGL би користили његов API на начин приказан на слици 3.1.2.

```
12  // Kreiramo skladište za objekat.
13
14  GLuint imeObjekta;
15  glGenObject(1, &imeObjekta);
16
17  // Smeštamo podatke u objekat.
18  glBindObject(GL_MODIFY, imeObjekta);
19  glObjectParameteri(GL_MODIFY, GL_OBJECT_COUNT, 10);
20  glObjectParameterf(GL_MODIFY, GL_OBJECT_OPACITY, 0.66f);
21  glObjectParameters(GL_MODIFY, GL_OBJECT_NAME, "Neki tekst.");
```

Слика 3.1.2. - Апстракција OpenGL позива

Наравно, ово нису стварни OpenGL позиви. Ово је једноставан пример која је разлика у интерфејсу.OpenGL задржава складиште свих својих објеката. Због овога, корисник може приступити објекту само по референци. Сви објекти су идентификовани користећи тип **unsigned int** - дефиниција типа за **GLuint**. Објекти су креирани пут функције имена **glGenType**, где **Type** представља тип објекта који се креира. Први аргумент је број објеката који ће бити креирани, док је други аргумент низ типа **GLuint** где ће бити смештене идентификације новокреираних објеката.

Да би изменили објекте, морамо их прво закачити за контекст. Објекти могу бити закачени за различите локације у контексту; ово дозвољава да један објекат буде искоришћен на различите начине. Поменуте различите локације се називају циљеви; сви објекти имају листу валидних циљева, а неки имају само један циљ. У претходном кодном примеру, фиктивни циљ GL_MODIFY је локација за коју је imeObjekta закачен. Можемо сматрати циљ као глобални показивач. У нашем случају, циљ GL_MODIFY је само глобални показивач који може чувати објекат овог типа (слика 3.1.3).

```
25  // Позивање glBindObject(GL_MODIFY, imeObjekta) би било исто што и:
26  Objekat* GL_MODIFY = NULL;
27
28  // glBindObject();
29  GL_MODIFY = ptr(imeObjekta);
```

Слика 3.1.3. - Објашњење функције glBindObject()

Функције које измењују ове објекте само измењују објекте који су закачени за контекст. Те функције примају, као први аргумент, циљ објекта који је намењен за измену. Ово референцира један од закачених објеката.

Енумератори GL_OBJECT_* набрајају сва својства која се могу сетовати у објекту. Породица функција glObjectParameter (слика 3.1.4) сетује параметре унутар објекта који је закачен за задати циљ. Како је OpenGL C API, свака варијација типа мора бити засебно именована функција. Тако је ту glObjectParameteri за целобројне аргументе, glObjectParameterf за аргумент реалног броја и тако даље.

```
35  // Када је у питању код, о овим функцијама можете мислити
36  // на следећи начин.
37
38  // glObjectParameteri(GL_MODIFY, GL_OBJECT_COUNT, 10);
39  GL_MODIFY->broj = 10;
40
41  // glObjectParameterf(GL_MODIFY, GL_OBJECT_OPACITY, 0.66f);
42  GL_MODIFY->prozirnost = 0.66f;
```

Слика 3.1.4. - Објашњење фамилије функција glObjectParameter()

Качење нуле за циљ у контексту је еквивалент сетовања циљовог глобалног показивача на NULL - откачиње који год објекат је био закачен за тај циљ.

OpenGL објекти нису једноставни као у овом примеру због функција које мењају стање објеката - не прати се конвенција именовања.

3.2. СПЕЦИФИКАЦИЈА И ИМПЛЕМЕНТАЦИЈА

Технички, OpenGL није API; већ спефицикација - документ. С API је један од начина за имплементацију спецификације. Спецификација дефинише иницијално стање OpenGL-a, шта која функција ради да би променила или примила то стање и шта би требало да се догоди када се позове функција за рендеровање.

Спецификација је написана од стране OpenGL архитектуалног ревизионог одбора (ARB), групе представника из великих технолошких компанија као што су Apple, NVIDIA, AMD и део је Khronos групе (слика 3.2.1).

Слика 3.2.1. - Чланице Khronos групе

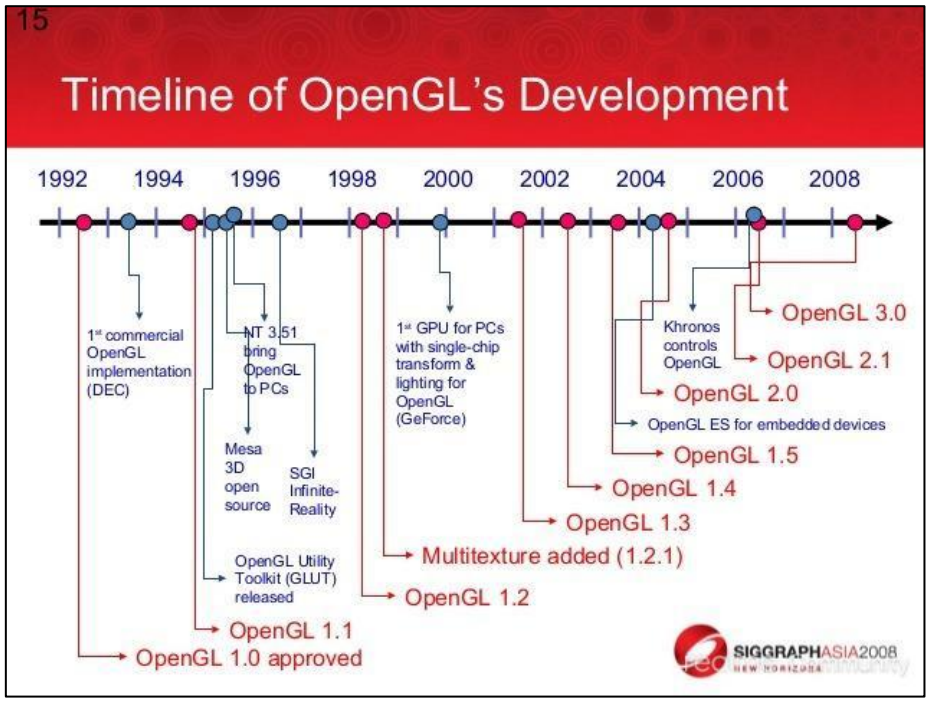
Спецификација је јако компликован и технички сложен документ. Делови тог документа су делимично разумљиви али је потребно основно познавање процеса. Уколико покушате да га прочитате, најважнија ствар је да се разуме следеће: **описује резултат, не имплементацију**. Ако спецификација каже да ће се ствар 1 догодити, то не значи да ће се ствар 1 заиста и догодити. То у ствари значи да корисник не би требало да види било какву разлику. Ако хардвер може пружити исто понашање на другачији начин, онда то спецификација дозвољава, докле год то корисник не може приметити.

Иако OpenGL ARB контролише садржај спецификације, не контролише OpenGL код. OpenGL није нешто што се може преузети са централизоване локације. За сваки засебан делић хардвера, посао програмера је да напишу OpenGL имплементацију за уређај на коме раде. Имплементација, као што и име каже, имплементира OpenGL спецификацију и излаже API као што је дефинисано у документу.

Контрола имплементације је различита за различите оперативне системе. На Windows оперативних системима, имплементација је у потпуности под контролом произвођача хардвера. На MAC OS X, имплементацију контролише Apple; они одлучују шта је изложено и каква додатна функционалност може бити пружена кориснику. Apple пише већину имплементације на MAC OS X, док произвођачи хардвера пишу за интерно креирани Apple API. На Linux оперативном систему ствари су сложене и ова тема то неће покрити.

Суштина приче о имплементацији и спецификацији је ако пишете програм и он демонстрира неочекивано понашање то је кривица креатора ваше OpenGL имплементације - уколико је са вашим кодом све у реду. На Windows оперативном систему, произвођачи графичких картица своју имплементацију уврштавају у своје управљачке програме. Прва ствар коју треба покушати је ажурирање графичких управљачких програма - можда је неправилност исправљена у најновијем ажурирању.

Постоји много верзија OpenGL спецификације. Верзије OpenGL-а нису као верзије Direct3D-а где се са сваком верзијом уводе агресивне промене API-ја. Код који функционише на једној верзији OpenGL-а ће функционисати на свакој будућој верзији (слика 3.2.2). Изузетак су верзије 3.0 и 3.1. Верзија 3.0 је прогласила застарелим број старијих функција док је верзија 3.1 избацила већину ових функција из API-ја. Ово је поделило спецификацију у два профила: језгро и компатибилност. Компатибилност задржава сву функционалност која је избачена у верзији 3.1, док језгро то не чини. Уколико би имплементација испоштовала језгро профил, ово би одбасцило сав софтвер који се ослања на профил компатибилности. У пракси, поменута подела је небитна. Ниједан програм неће доставити управљачке програме који имплементирају само језгро профил. У суштини, ово не значи ништа; све верзије OpenGL су уназад компатибилне. [26]



Слика 3.2.2. - Временска линија развоја OpenGL-а

3.3. ПРАКТИЧНИ ПРИМЕРИ

3.3.1. Креирање прозора

Пошто је OpenGL преносив на свим платформама, позиви за креирање прозора су специфични за платформу на којој радимо. Како би избегли непотребно куцање кода и задржали преносивост, користићемо библиотеке које ће те ствари обавити за нас.

Користићемо библиотеку GLFW (Graphics Library Framework) која је јако лагана и обавиће за нас само посао креирања прозора, креирање OpenGL контекста и корисничког уноса.

https://www.glfw.org/

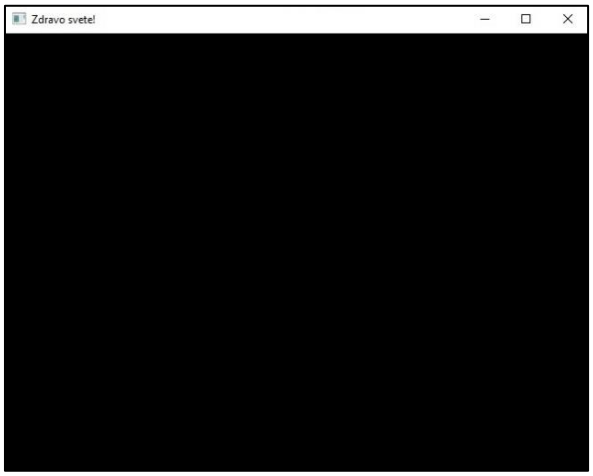
У овом завршном раду неће бити обухваћене тривијалне ствари као што су креирање пројекта, праћење документације, подешавање екстерних библиотека и подешавања **include** путања. За примере употребе користићемо **Visual Studio 2017 Community**.

```
10    /* Create a windowed mode window and its OpenGL context */
11    window = glfwCreateWindow(640, 480, "Zdravo svete!", NULL, NULL);
12    if (!window) {
13        glfwTerminate();
14        return -1;
15    }
```

Слика 3.3.1.1. - Креирање OpenGL прозора и контекстса

На кодном примеру на слици 3.3.1.1. смо користили исечак кода из GLFW документације и представља креирање прозора димезина 640x480 са насловом "**Zdravo svete!**".

Као што видимо на слици 3.3.1.2, резултат је скалирани празан прозор жељених димензија. То је и очекивано, пошто нисмо пружили никакве податке за исцртавање. Кренућемо од најједноставије графичке примитиве – троугла.



Слика 3.3.1.2. - Празан прозор

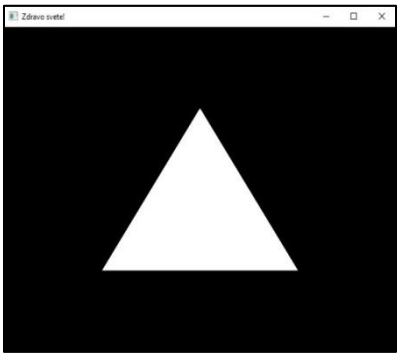
3.3.2. Исцртавање троугла – застарели (legacy) OpenGL

Тренутно имамо два начина за исцртавање троугла; коришћење застарелог OpenGL-а и модерног. Прво ћемо користити доста једноставнији начин – користићемо застарели OpenGL како би добили троугао на нашем прозору. Застарели (legacy) OpenGL се углавно састоји од предефинисаних стања.

```
25          // Назначујемо да желимо да исцртамо троугао.
26          glBegin(GL_TRIANGLES);
27
28          // Уписујемо три темена тог троугла типа float.
29          glVertex2f(-0.5f, -0.5f);
30          glVertex2f( 0.0f,  0.5f);
31          glVertex2f( 0.5f, -0.5f);
32
33          // Завршавамо OpenGL позив.
34          glEnd();
```

Слика 3.3.2.1. - Исцртавање троугла на застарели начин

Код на слици 3.3.2.1 се налази унутар основног кода из документације библиотеке GLFW.



Слика 3.3.2.2. – Резултат исцртавања

3.3.3. Употреба модерног OpenGL-а

За модерни OpenGL се сматра свака верзија новија од верзије 1.1. У примеру исцртавања троугла помоћу застарелог OpenGL-а (слика 3.3.2.2) смо користили позиве функција које пружа GLFW библиотека и Windows OpenGL библиотеке које иду само до верзије 1.1.

Да бисмо користили функционалности модерног OpenGL-а, потребно је да извучемо све модерне функције из графичких управљачких програма. Како је ово захтеван посао и зависан од платформе на којој радимо, користићемо још једну помоћну и јако лагану библиотеку – GLEW (OpenGL Extension Wrangler Library). Она обавља функцију достављања свих могућих OpenGL потписа функција и константи, затим проналази коју графичку картицу имамо и повезује њене библиотеке са нашим потписима функција (слика 3.3.3.1).

http://glew.sourceforge.net/

```
19      /* Make the window's context current */
20      glfwMakeContextCurrent(window);
21
22      // Иницијализација GLEW библиотеке.
23      GLenum err = glewInit();
24
25      if (GLEW_OK != err) {
26          // Провера успешности иницијализације.
27          std::cout << "Greska: " << glewGetErrorString(err) << std::endl;
28      }
29      // Испис тренутне верзије GLEW-а коју користимо.
30      std::cout << "Status: Using GLEW " << glewGetString(GLEW_VERSION) << std::endl;
```

Слика 3.3.3.1. - Иницијализација GLEW библиотеке

Пратећи документацију долазимо до информације да је иницијализација библиотеке могућа само након креирања OpenGL контекста. [27]

3.3.4. Исцртавање троугла – модерни OpenGL

Да би смо били у могућности да исцртамо троугао на екрану, потребе су нам две ствари: бафер темена и шејдер. Бафер темена није ништа друго но бафер који се налази на видео меморији и припада меморији OpenGL-а. У следећем примеру ћемо дефинисати податке које описују наш троугао, то ћемо проследити у видео меморију и затим ћемо иницирати позив исцртавања.

Међутим, да би позив исцртавања био успешан, морамо послати детаљне инструкције графичкој картици како ми желимо да она интерпретира податке које смо јој послали. Све што напишемо на C++-у се извршава на процесору – када процесор изврши наше инструкције ток извршавања прелази на графичку картицу. Да би смо упутили нашу графичку картицу шта да ради са нашим обрађеним подацима морамо написати **shader** (шејдер). Шејдер је ништа друго но програм који се изршава на графичкој картици. У овом случају, проследићемо гомилу података графичкој картици, затим ћемо јој помоћу шејдера саопштити да исцрта три темена и да их споји како би добили троугао.

Потребно је да одредимо позиције наших темена, у овом случају имаћемо три темена од којих ће свако теме бити представљено кроз X,Y координате. То су: [-0.5 , -0.5] лево теме, [0 , 0.5] горње теме и [0.5 , -0.5] десно теме (слика 3.3.4.1.).

```
32    // Низ са позицијама наших темена.
33    float pozicije[6] = {
34        -0.5f, -0.5f,
35         0.0f,  0.5f,
36         0.5f, -0.5f
37    };
38
39    // Идентификација нашег бафера (сваки објекат у OpenGL је има)
40    unsigned int baferID;
41    // Генеришемо један бафер и његову идентификацију смештамо у baferID
42    glGenBuffers(1, &baferID);
43    // Наш бафер качимо за контекст
44    glBindBuffer(GL_ARRAY_BUFFER, baferID);
45    // Пунимо наш бафер са подацима из низа позиција.
46    glBufferData(GL_ARRAY_BUFFER, 6 * sizeof(float), pozicije, GL_STATIC_DRAW);
```

Слика 3.3.4.1. - Уписивање позицијама темена и пуњење бафера

Уколико сада позовемо фукнцију за исцртавање, наш прозор ће остати празан. Ово није случај код неких графичких управљачких програма – неки имају основни шејдер за потребе отклањања грешака или спречавања пуцања програма, свакако спада у недефинисано понашање и идемо са претпоставком да нема никаквих подразумеваних шејдера. Потребно је да напишемо одговарајући шејдер који ће знати шта да ради са бафером којег смо сместили у видео меморију. Након тога неопходно је да дефинишемо атрибуте наших темена (vertex attributes) и да одредимо распоред меморије (layout). [28]

Атрибути темена могу бити позиција, координате текстуре, нормале, боје итд. Самим тим једно теме није дефинисано само позицијом (у нашем једноставном случају за сада јесте). Распоред меморије је неопходно одредити како би шејдер знао колика је величина у бајтовима за одређени атрибут као и који је офсет сваког засебног атрибута.

На пример, у случају дефинисања распореда за наш бафер троугла потребно је шејдеру експлицитно рећи да је величина једне позиције темена 8 бајтова – две величине типа **float** и да меморија позиција почиње од самог почетка, да не постоји никакав офсет (слика 3.3.4.2).

```
48      // Омогућавамо атрибут темена са индексом 0 (позиција)
49      glEnableVertexAttribArray(0);
50      // Дефинишемо распоред меморије нашег бафера - атрибут величине два float типа.
51      glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 2 * sizeof(float), 0);
```

Слика 3.3.4.2. - Омогућавање атрибута темена и дефиниција меморије бафера

Добар извор документације о OpenGL API би био веб сајт:

https://docs.gl/

Претрага се врши по функцији и постоји детаљан опис сваке функције по верзији OpenGL-а – спецификација, параметри, опис, употреба и могуће грешке.

type

Specifies the data type of each component in the array. The symbolic constants `GL_BYTE`, `GL_UNSIGNED_BYTE`, `GL_SHORT`, `GL_UNSIGNED_SHORT`, `GL_INT`, and `GL_UNSIGNED_INT` are accepted by `glVertexAttribPointer` and `glVertexAttribIPointer`. Additionally `GL_HALF_FLOAT`, `GL_FLOAT`, `GL_DOUBLE`, `GL_FIXED`, `GL_INT_2_10_10_10_REV`, `GL_UNSIGNED_INT_2_10_10_10_REV` and `GL_UNSIGNED_INT_10F_11F_11F_REV` are accepted by `glVertexAttribPointer`. `GL_DOUBLE` is also accepted by `glVertexAttribLPointer` and is the only token accepted by the *type* parameter for that function. The initial value is `GL_FLOAT`.

Слика 3.3.4.3. - Пример документоване функције

На слици 3.3.4.3 је приказан опис трећег аргумента функције **glVertexAttribPointer()**.Након што смо дефинисали атрибут темена и распоред меморије бафера, следећи корак је да напишемо одговарајући шејдер.

Постоје два типа шејдера који се највише користе: теме шејдер (vertex shader) и фрагмент шејдер (vertex shader или пиксел шејдер). Када позовемо функцију за исцртавање на екрану, прво ће се позвати теме шејдер, затим фрагмент шејдер и након њихове обраде података видећемо резултат исцртан. Иако је процес доста упрошћен, сви корици које смо преузели до сад представљају део графичког тока (pipeline). Теме шејдер ће бити позван за свако теме – у нашем случају три пута. Употреба теме шејдера је указивање OpenGL-у где желимо наша темена на простору прозора наше апликације (screen space). Такође се користи за даље прослеђивање података – у нашем случају фрагмент шејдеру. Фрагмент шејдер ће бити позван за сваки пиксел којег је потребно исцртати на екрану – његова улога је да за сваки пиксел израчуна боју којом ће бити исцртан. Попуњавање примитиве коју желимо да исцртамо – у овом случају троугла – је јако слично раду екрана са катодном цеви поменутог раније у овом раду.

Пример: имамо квадрат величине 200 x 200 пиксела. Желимо да извршимо неко израчунавање – на пример X * Y. Уколико то урадимо у теме шејдеру, то израчунавање ће се извршити 4 пута (за свако теме). Уколико то урадимо у фрагмент шејдеру, то ће се извршити 40.000 пута – јако скупо и могуће беспотребно трошење ресурса! [29]

Да би написали шејдер, прво су нам потребне две функције како би избегли код који се понавља и исти је за оба типа шејдера. Прва функција ће креирати OpenGL програм спајањем два компајлирана шејдера (слика 3.3.4.4) путем друге функције која ће креирати шејдер, напунити га изворним кодом и компајлирати а затим вратити његову идентификацију (слика 3.3.4.5). Повратна вредност главне функције је идентификација програма како би га касније закачили за контекст.

```
41  // Функција прима као аргументе два изворна кода у виду стрингова.
42  GLuint KreirajProgram (const std::string& temeSejderKod, const std::string& fragmentSejderKod) {
43      // Креирамо OpenGL програм и чувамо његову идентификацију.
44      GLuint programID = glCreateProgram();
45      // Компајлујемо теме шејдера и чувамо његову идентификацију.
46      GLuint tsID = KompajlujSejder(GL_VERTEX_SHADER, temeSejderKod);
47      // Компајлујемо фрагмент шејдер и чувамо његову идентификацију.
48      GLuint vsID = KompajlujSejder(GL_FRAGMENT_SHADER, fragmentSejderKod);
49
50      // Качимо шејдере за OpenGL програм.
51      glAttachShader(programID, tsID);
52      glAttachShader(programID, vsID);
53      glLinkProgram(programID);
54      glValidateProgram(programID);
55
56      // Бришемо непосредне фајлове који су настали у процесу.
57      glDeleteShader(tsID);
58      glDeleteShader(vsID);
59
60      return programID;
61  }
```

Слика 3.3.4.4. - Креирање OpenGL програма

Како би били сигурни да је све прошло у најбољем реду, написаћемо код који ће проверити статус компилације шејдера. Провера грешака у OpenGL-у је јако робусна и генеричка за већину провера (генеричка у смислу да су кораци исти али су позивне функције и дохватање података различити). Интересантно је да се у нашој функцији налази више кода за проверу грешака него кода који компајлује шејдер.

```
5   GLuint KompajlujSejder(unsigned int tip, const std::string& izvorniKod) {
6       // Креирамо шејдер и чувамо његову идентификацију.
7       GLuint idSejdera = glCreateShader(tip);
8       // Кастујемо C++ стринг у C стринг како би могли да га проследимо OpenGL функцији.
9       const char* izvorniKodC = izvorniKod.c_str();
10      // Пунимо наш новокреирани шејдер са спољним изворним кодом.
11      glShaderSource(idSejdera, 1, &izvorniKodC, NULL);
12      // Компајлујемо шејдер.
13      glCompileShader(idSejdera);
```

Слика 3.3.4.5. - Компајлирање шејдера

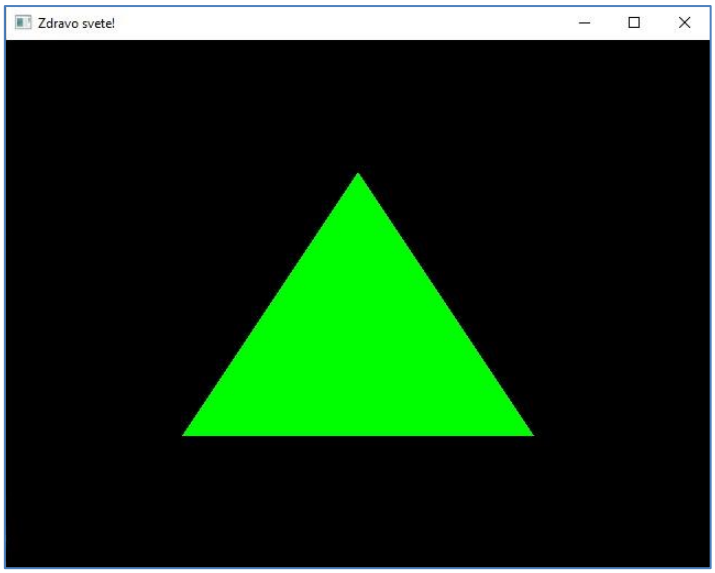
Сада када имамо функције које ће нам доставити готов OpenGL програм можемо приступити писању изворног кода за два типа шејдера (слика 3.3.4.6).

```
111    std::string temeSejderKod = R"glsl(
112        #version 330 core
113        layout(location = 0) in vec4 position;
114        void main(){
115            gl_Position = position;
116        }
117    )glsl";
118
119    std::string fragmentSejderKod = R"glsl(
120        #version 330 core
121        layout(location = 0) out vec4 color;
122        void main(){
123            color = vec4(0.0, 1.0, 0.0, 1.0);
124        }
125    )glsl";
126
127    GLuint sejderProgram = KreirajProgram(temeSejderKod, fragmentSejderKod);
128    glUseProgram(sejderProgram);
```

Слика 3.3.4.6. - Изворни кодови шејдера

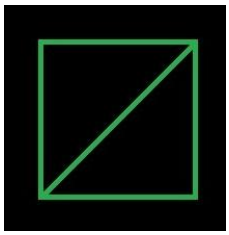
Као што на слици 3.3.4.7, наши шејдери су веома прости али обављају посао. Назнака **#version 330 core** је инструкција OpenGL-у коју ћемо верзију GLSL-а (**OpenGL Shading Language**) користити – у овом случају корену верзију 3.3.0 (без застарелих функција). Након тога, из бафера се извлаче координате темена по пређашњем дефинисаном меморијском распореду (ствари постају много јасније када се процес употпуни). Позиција темена се сетује на спољну позицију, и шејдер темена се позива за свако теме – у овом случају три пута.

У фрагмент шејдеру дешава се супротно – боја се одређује и шаље у атрибуте темена. Изабрана боја је (**0.0 , 1.0 , 0.0 , 1.0)** – како је формат RGBA добићемо зелену боју (вредности од 0 до 255 се нормализују у вредности од 0 до 1). Не треба заборавити да се фрагмент шејдер позива једном по пикселу – уколико је наш троугао површине 1000 пиксела – толико ће позива уследити. [30]



Слика 3.3.4.7. - Резултат досадашњег рада

3.3.5. Исцртавање квадрата



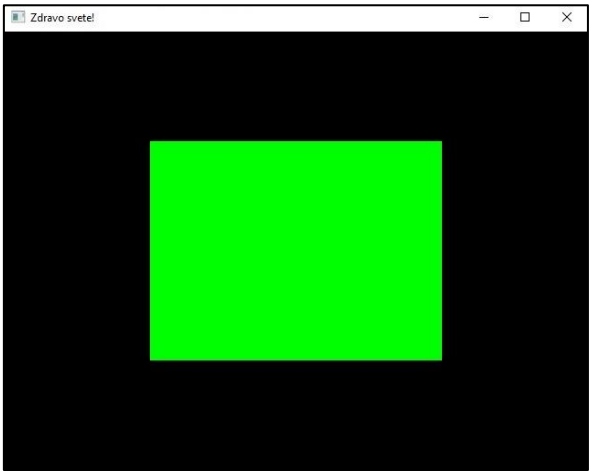
Слика 3.3.5.1. - Састављање квадрата из два троугла

Као што смо раније поменули, троугао је најједноставнија примитива са најмањим бројем темена за репрезентацију равни чија нормала на површ има само један смер. Самим тим, да би добили квадрат, једноставно ћемо нацртати два троугла (слика 3.3.5.1).

Потребно је да благо изменимо постојећи код како би добили два троугла као на слици. Прво ћемо изменити координате темена постојећег троугла како би добили десни, а како имамо само три темена, додаћемо још толико за репрезентацију левог троугла (слика 3.3.5.2). Уз ситне измене као што су колико бајтова алоцирати за графички бафер и промену броја темена резултат се види на слици 3.3.5.3.

```
90    float pozicije[] = {
91    // Десни троугао.
92        -0.5f, -0.5f,
93         0.5f, -0.5f,
94         0.5f,  0.5f,
95
96    // Леви троугао.
97         0.5f,  0.5f,
98        -0.5f,  0.5f,
99        -0.5f, -0.5f
100   };
```

Слика 3.3.5.2. - Координате два троугла



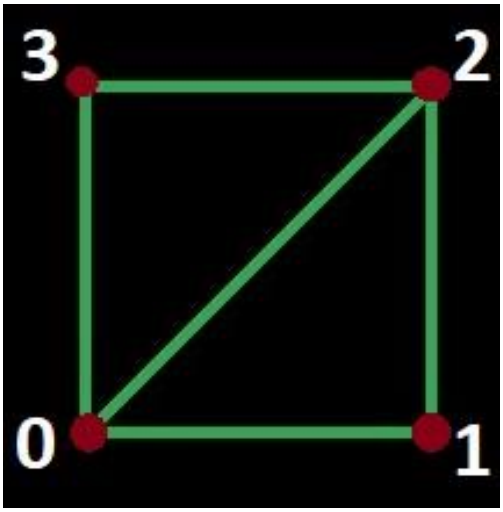
Слика 3.3.5.3. - Резултат исцртавања

Можемо приметити да нисмо добили квадрат већ правоугаоник али је то због димензија нашег прозора (640x480). Међутим, можемо приметити један проблем који у овом случају не делује озбиљно. Анализирамо леви троугао, тачка у доњем левом углу је тачка А, тачка у горњем левом углу тачка Б и тачка у доњем десном углу тачка Ц. Настављамо са десним, тачка Д у доњем десном углу, тачка Е у горњем десном углу и тачка Ф у горњем левом углу. Изводимо закључак да су координате два различита темена исте (горње лево и доње десно). Како су координате у програму записане помоћу типа **float** (величина на тестираном систему 4 бајта) ми тренутно беспотребно користимо 16 бајтова. У нашем случају то није озбиљан проблем оптимизације и можемо га занемарити, али уколико би се тај проблем појавио у **game engine**-у помоћу ког је направљен огроман модел свемирског брода који је састављен од хиљада и хиљада троуглова, губитак ресурса због неозбиљног приступа оптимизацији би био у мегабајтима који су могли бити боље искоришћени (на пример за текстуру високе дефиниције).

Начин на који би могли да уштедимо поменутих 16 бајтова је коришћено **индекс бафера**. То је бафер који нам дозвољава да једно теме користимо више пута. Једноставно ћемо у том баферу записати распоред четири темена (слика 3.3.5.5) са могућношћу поновне употребе.

```
90    float pozicije[] = {
91        -0.5f, -0.5f, // 0
92         0.5f, -0.5f, // 1
93         0.5f,  0.5f, // 2
94        -0.5f,  0.5f  // 3
95    };
96
97    unsigned int indeksi[] = {
98        0, 1, 2,
99        2, 3, 0
100   };
```

Слика 3.3.5.4. - Низови позиција и индекса



Слика 3.3.5.5. - Редослед исцртавања

Као што видимо на слици 3.3.5.4, два темена која су се понављала смо избацили, и додали смо низ типа **unsigned int** са вредностима који означавају редослед темена наших два троуга бирајући темена која се понављају у низу позиција. Сада морамо индекс бафер послати из централне процесорске јединице у графичку процесорску јединицу (слика 3.3.5.6). Процес је сличан креирању бафера ког смо искористили за складиштење позиција темена.

```
120   // Идентификација нашег другог бафера.
121   GLuint indeksBaferID;
122   // Генеришемо један бафер и смештамо његову идентификацију у променљиву.
123   glGenBuffers(1, &indeksBaferID);
124   // Нови бафер качимо за контекст - први се аутоматски откачи.
125   glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indeksBaferID);
126   // Пунимо нови бафер са подацима из низа индекса.
127   glBufferData(GL_ELEMENT_ARRAY_BUFFER, 6 * sizeof(GLuint), indeksi, GL_STATIC_DRAW);
```

Слика 3.3.5.6. - Креирање индекс бафера и пуњење подацима

```
154       // glDrawArrays(GL_TRIANGLES, 0, 6);
155       // Промена позива за исцртавање због коришћења индекс бафера.
156       glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, NULL);
```

Слика 3.3.5.7. – Измењени позив исцртавања

Уштедели смо меморију и исцртали квадрат на најоптималнији могући начин (слика 3.3.5.7) – визуелно квадрат изгледа исто као и претходни. Међутим, индекс бафери по спецификацији морају бити типа **unsigned int (проширени скуп природних бројева)**, шта да смо ми случајно као почетници грешком уписали тип **int** а затим као један од чланова низа индекса уписали **-1**? Који је **-1** индекс? OpenGL механизми за отклањање и јављање грешака су јако „сувопарни“ – у овој ситуацији би добили црн екран без икаквог саопштења шта није у реду. [31]

3.3.6. Отклањање грешака

Нећемо користити екстерне алате за анализу нашег OpenGL програма већ ћемо се ослонити на интерне механизме за отклањање грешака. Постоји два начина за дохватање грешака:

- **glGetError() -** функција која је присутна у спецификацији од првих верзија. Сваки пут када се догоди нека грешка сетује се интерни флег (**flag - заставица**) у меморији OpenGL-а. Ова функција ће нам дохватити тај флег који је у основи тип **int** односно добићемо код грешке. Уколико направимо више грешака, потребно је више пута за редом позвати ову функцију јер један позив дохвата само један флег (флегови се не копирају већ се преносе из OpenGL меморије – ова функција самим тим и ресетује стање флегова!). Принцип како функционише отклање грешака уз помоћ ове функције је једноставан али и мало кабаст – пре сваког позива OpenGL функцији прво ћемо уз помоћ **while** петље и ове функције ресетовати стање флегова, затим ћемо позвати неку функцију (на пример **glDrawElements())** и одмах након тога поновна провера да ли је промењен неки од флегова – уколико јесте, одмах можемо открити која функција је узроковала грешку. Овај начин представља најједноставнији и најчешће употребљени јер је компатибилан са свим верзијама спецификације.

- **glDebugMessageCallback() –** нова функција за отклањање грешака која је присутна тек од верзије 4.3. Као аргумент прима показивач на функцију коју ће OpenGL аутоматски позвати када наступи грешка, самим тим не морамо проверавати флегове после сваког позива функције. Још једна предност је што су поруке за отклањање много детаљније – добија се тип грешке, идентификација, важност као и порука о грешци.

Данас ћемо обрадити развијање механизама за отклањање грешака уз помоћ функције **glGetError()**. Позив функције враћа код грешке типа **GLenum** и затим се флегови ресетују на вредност **GL_NO_ERROR**. Уколико функција врати подразумевану вредност, то значи да ниједна грешка није откривена од последњег позива функције. Због својства функције да враћа једну арбитраран код грешке, препоручује се пре позива ресетовање флегова (слика 3.3.6.1).

```
5   // Функција за ресетовања флегова
6   static void GLObrisiGresku() {
7       // Можемо написати и као !glGetError()
8       while (glGetError() != GL_NO_ERROR);
9   }
10
11  static void GLProveriGresku() {
12      while (GLenum greska = glGetError())
13          std::cout << "[OpenGL Greska] (0x" << std::hex << greska << ")" << std::endl;
14  }
```

Слика 3.3.6.1. - Функције за проверу и брисање OpenGL грешака

Дефинисане су следеће грешке:

- **GL_NO_ERROR – 0x0000 –** означава да није грешка није регистрована. Вредност ове симболичке константе је гарантована да је 0.
- **GL_INVALID_ENUM – 0x0500 –** неадекватна вредност је пружена као енумеровани агрумент. Проблематична функција је игнорисана и нема споредног ефекта сем сетовања заставице грешке.
- **GL_INVALID_VALUE – 0x0501** – нумерачки аргумент је ван опсега. Проблематична функција је игнорисана и нема споредног ефекта сем сетовања заставице грешке.
- **GL_INVALID_OPERATION – 0x0502** – затражена операција није валидна у тренутном контексту извршавања. Проблематична функција је игнорисана и нема споредног ефекта сем сетовања заставице грешке.
- **GL_INVALID_FRAMEBUFFER_OPERATION – 0x0506** – бафер оквира објекта није комплетан. Проблематична функција је игнорисана и нема споредног ефекта сем сетовања заставице грешке.
- **GL_OUT_OF_MEMORY – 0x0505** – нема довољно преостале меморије за извршавање наредбе. Стање извршавања је недефинисано, сем стања заставица за грешке.
- **GL_STACK_UNDERFLOW – 0x0504** – покушана је операција која би довела до интерног подлива стека (**stack underflow**).
- **GL_STACK_OVERFLOW – 0x0503** – покушана је операција која би довела до интерног прелива стека (**stack overflow**).
- **GL_CONTEXT_LOST** – **0x0507 –** сетована уколико је целокупни контекст изгубљен због ресетовања графичке картице (тек од верзије 4.5 и као повратна вредност функције **glGetGraphicsResetStatus()**. [32]

На слици 3.3.6.2 се види употреба механизма пре и после позива функције **glDrawElements()**. Резултат извршавања проблематичног кода је приказан на слици 3.3.6.3. По табели изнад можемо видети да смо начинили грешку типа **GL_INVALID_ENUM.**

```
166        GLObrisiGresku();
167        glDrawElements(GL_TRIANGLES, 6, GL_INT, NULL);
168        GLProveriGresku();
```

Слика 3.3.6.2. - Употреба механизма у коду

```
[OpenGL Greska] (0x0500)
```

Слика 3.3.6.3. - Исписани код грешке

Усвојени приступ је мало незграпан, имајући у виду да ми пре сваке OpenGL функције морамо да позовемо две функције, за брисање и исписивање грешака. Можемо одредити у ком делу програма се догодила грешка простом опсервацијом – у досадашњем случају грешка се исписује сваки фрејм самим тим закључујемо да се грешка догодила у петљи за исцртавање. Уколико имамо озбиљан OpenGL програм са хиљадама линија кода, отклањање грешака овим механизном постаје јако незахвално јер не можемо одредити на којој линији је дошло до грешке. Постојећи механизам можемо утврдити коришћењем уграђених алата нашег окружења за развијање.

ASSERT механизам (слика 3.3.6.4) прекида извршавање програма тачно на линији кода која није прошла проверу. Еквивалент би био ручно уношење тачака за прекид (**breakpoint**) у режиму рада за отклањање грешака (**debug mode)**. Важно је напоменути да ако покренемо програм у режиму за издавање (**release mode**) велика је вероватноћа да можда до наше провере неће ни доћи због оптимизације компајлера. Како користимо развојно окружење **Visual Studio**, користићемо функцију **__debugbreak()** која ће уместо нас аутоматски прекинути извршавање програма на линији која није прошла провера и поставити тачку за прекид (слика 3.3.6.7). Коришћење интерних функција компајлера се назива интристика компајлера и функције су различите од компајлера до компајлера – што би значило да се наш код неће компајлирати уколико користимо други компајлер – на пример **GCC/G++.**

```
5   #define ASSERT(x) if(!(x)) __debugbreak();
```

Слика 3.3.6.4 - Претпроцесорска директива за дефиницију ASSERT механизма

Функцију **GLProveriGresku()** смо преименовали у **GLPozivProvere()** јер смо је изменили да враћа вредност типа **boolean – true** уколико је дошло до грешке. Овим додатком смо решили проблем незнања линије кода – преостаје да напишемо одговарајући макро који ће ова два позива функције уједињити у један и махом аутоматизовати процес отклањања грешака (слике 3.3.6.5 и 3.3.6.6).

```
7   #define GLPoziv(x) GLObrisiGresku();\
8                      x;\
9                      ASSERT(GLPozivProvere());
```

Слика 3.3.6.5. - Претпроцесорска директива за дефиницију макроа GLPoziv()

```
177     GLPoziv(glDrawElements(GL_TRIANGLES, 6, GL_INT, NULL));
```

Слика 3.3.6.6. - Употреба макроа GLPoziv()

```
178     GLPoziv(glDrawElements(GL_TRIANGLES, 6, GL_INT, NULL));
179
180     /* Swap front and back buffers */
181     glfwSwapBuffers(window);
182
```

Exception Thrown

OpenGL.exe has triggered a breakpoint.

Слика 3.3.6.7. - Прекид извршавања програма на линији 178

Даље развијање нашег главног позива за отклањање грешака је могуће уз прослеђивање имена функције, фајла у ком је наступила грешка и компајлерску функцију **__LINE__** која ће на открити на којој линији се догодио прекид. [33]

4. ГЕНЕРИСАЊЕ СПЛАЈНОВА

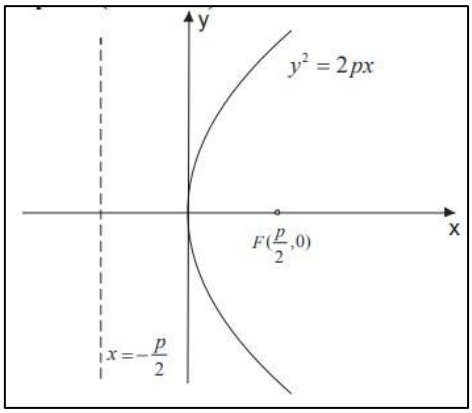
4.1. ТЕОРИЈА

4.1.1. Крива

Крива је геометријски појам, апстракција обичне представе криве линије. У различитим сферама математике термин крива се дефинише на различите начине, зависно од циљева и метода изучавања. Крива је геометријско место тачака простора чије су координате функције једно променљиве. Различите дефиниције криве захтевају различиту глаткоћу ових функција. Крива у Жордановом смислу задаје се непрекидним функцијама и може веома да се разликује од обичне представе криве линије. Ректификабилна крива има координате тачака које су непрекидно диференцијабилне функције параметара. Код овакве криве се може увести појам дужина лука. Алгебарске криве се задају једначинама: $F_1(x,y,z)=0, F_2(x,y,z)=0$, где су F_1,F_2 полиноми од три променљиве. Права је алгебарска крива. [34]

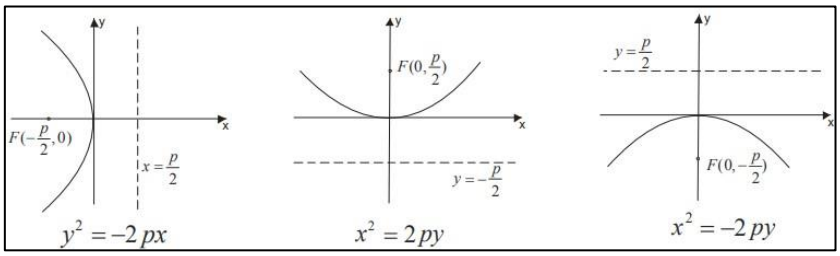
4.1.2. Парабола

Парабола је скуп тачака у равни са осовином да је растојање сваке тачке од једне сталне тачке (жиже) једнако одстојању те тачке од једне сталне праве (директрисе).



Слика 4.1.2.1. - Парабола једначине $\boldsymbol{y^2 = 2px}$

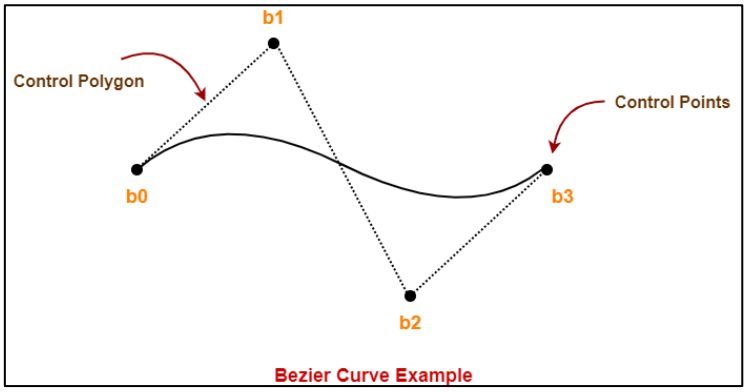
$F\left(\frac{p}{2},0\right)$ је жижа параболе. Права $x = -\frac{p}{2}$ је директриса параболе или $x+\frac{p}{2}=0$. Одстајање тачке F од директрисе обележава се са $\boldsymbol{p}$ и назива се параметар параболе. Координатни почетак је теме параболе. Једначина параболе је $y^2 = 2px$ (слика 4.1.2.1). Мање заступљене параболе су приказане на слици 4.1.2.2. [35]



Слика 4.1.2.2. – Мање заступљене параболе

4.1.3. Безјеове криве

Безјеове криве су параметарске криве. Задају се одређеним бројем контролних тачака чијим се померањем мења облик криве (слика 4.1.3.1). Додавањем контролних тачака се повећава степен криве, али и сложеност израчунавања због чега се најчешће користе криве малог степена. „Лепљењем“ кривих малог степена, добијају се Безјеови сплајнови (сложене криве). Пол Кастелжи и Пјер Безје су их при користили при дизајну аутомобила крајем педесетих година прошлог века.



Слика 4.1.3.1. - Контролне тачке и контролни полигон Безјеове криве

Рекурзивна дефиниција:

Ако Безјеову криву одређену помоћу тачки $P_0, P_1, P_2 .. P_n$ означимо као $\alpha_{P0P1 \dots Pn}$ онда:

$$\propto_{P0} (t) = P_0 i$$

$$\propto_{P0} (t) = \propto P_0 P_1 \dots P_n(t) = (1 - t) \propto P_0 P_1 \dots P_{n-1}(t) + t \propto P_1 P_2 .. P_n(t)$$

Експлицитна дефиниција:

Нека је Безјеова крива степена $n \geq 2$, задата помоћу $P_0, P_1, P_2 .. P_n, n \geq 2$ тачака равни. Можемо је представити следећим полиномом:

$$\propto (t) = \sum_{i=0}^{n} (\frac{n}{i})\, t^i\, (1 - t)^{n-i} P_i = \sum_{i=0}^{n} B_i^n\, (t)\, P_i, t \in [0,1]$$

Тачке P_i помоћу који је дефинисана крива називају се контролне тачке, а полиноми B_i^n Берштајнови полиноми тј. базне функције. Тачке $P_0, P_1, P_2 .. P_n$ образују контролни полигон.

Полиномијални облик:

Применом биномне теореме на дефиницију криве добијамо:

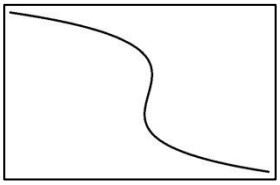
$$\propto (t) = \sum_{j=0}^{n} t^j C_j$$

где је:

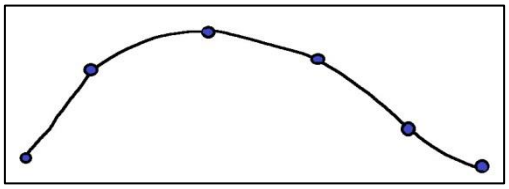
$$\propto' (t) = n \sum_{i=0}^{n-1} b_{i,n-1}\, (t)(P_{i+1} - P_i)$$ [36]

4.2. КАТМУЛ-РОМ СПЛАЈН

Сплајн је начин за алгоритамско представљање криве. Сплајнови могу бити од велике користи. На пример, у видео играма непријатељи се крећу напред назад по правој линији. Употребом сплајнова, могли бисмо да дефинишемо криву по којој ће се они кретати и њихово кретање ће изгледати доста природније. Сплајнови имају две контролне тачке и трећу, која се користи као индикација смера нагињања криве. На слици 4.2.1. се може видети једноставан сплајн из програма Paint.



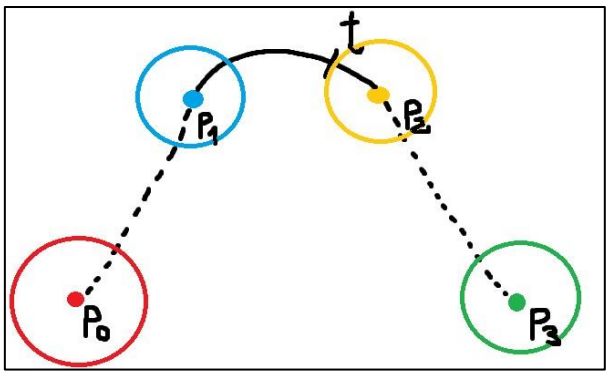
Слика 4.2.1. - Једноставан сплајн



Слика 4.2.2. - Сплајн са загарантованим пролазом кроз све контролне тачке

Уколико покушамо да овај модел искористимо да продужимо путању крива, мораћемо да погађамо где би контролне тачке требало да се налазе. Исто тако се јављају проблеми када покушамо да спојимо два сплајна, такође математика иза сплајова је поприлично озбиљна. Из ових разлога користићемо Катмул-Ром сплајнове који користе више контролних тачака и загарантован је пролазак криве кроз све тачке (слика 4.2.2).

У основној форми, Катмул-Ром сплајн сачињавају четири тачке (P0,P1,P2,P3). Катмул-Ром алгоритам ће генерисати криву само између тачака P1 и P2 док ће се тачке P0 и P3 користити као смернице угла криве у пролазу кроз тачке P1 и P2. Да би се пронашла тачка на сплајну, углавном се користи вредност t која почиње од вредности 0 на тачки P1 и завршава се вредношћу 1 на тачки P2 (приступамо сегменту криве у нормализованом простору). Предност Катмул-Ром сплајна је што тачке могу бити вишедимензионалне, притом се само рачуница скалира. Да би схватили како сплајн функционише. треба мислити на тачке P1 и P2 као на центре утицајних кружница које воде нашу тачку t. Што смо ближи тачки P2, већи је утицај тачке P2 и самим тим опада утицај тачке P1. Уколико је тачка t тачно на средини подједнако је под утицајем обе тачке. Уколико се тачка t налази тачно на тачки P1, она је у потпуности под утицајем само тачке P1. Улога кружница око тачака P0 и P3 је да одбијају нашу тачку али не толико драстично као што је утицај P1 и P2 (слика 4.2.3).



Слика 4.2.3. - Утицајне кружнице у Катмул-Ром сплајну

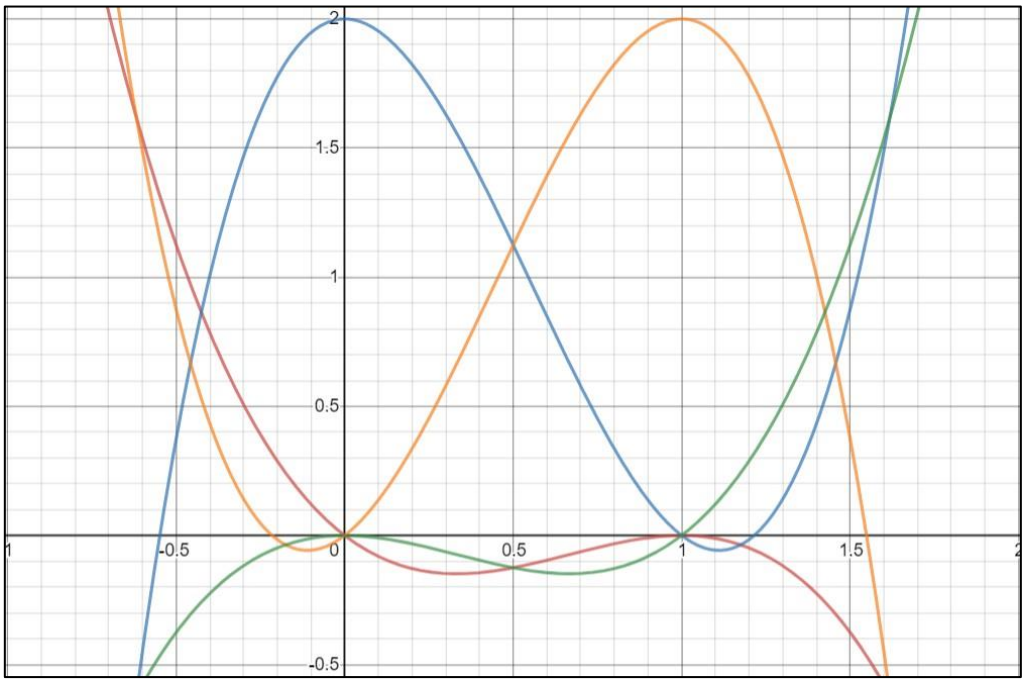
Графици функције (слика 4.2.4) нам могу помоћи да употпунимо разумевање функционисања Катмул-Ром сплајнова. Посматрајмо X осу као репрезентацију наше тачке t у нормализованом простору. Утицај тачке P1 се огледа у плавој функцији пресликаној на у оси која приказује њен утицај на тачку t у зависности од близине. У тачки { 0, 2 } тачка t је под потпуним утицајем тачке P1, док је тај утицај у тачки {1, 0} раван нули. Утицај тачке P2 је у супротности са деловањем тачке P1. Све ове функције су кубне, самим тим ово је кубни сплајн.

$$y_1 = -x^3 + 2x^2 - x$$

$$y_2 = 3x^3 - 5x^2 + 2$$

$$y_3 = -3x^3 + 4x^2 + x$$

$$y_4 = x^3 - x^2$$



Слика 4.2.4. - Деловање сваке кружнице приказано на графикуфункције

Графицима функције се може приступити на следећем линку:

https://www.desmos.com/calculator/q6lzef5du6

Једна од мана Катмул-Ром сплајна је што не можемо одредити градијент кроз тачку. Градијент је увек израчунат помоћу позиције његових суседних тачака. Ови изрази су валидни само за једну компоненту вектора наше тачке. У дводимензионалном простору, морамо да израчунамо све ове функције за X компоненту, а затим за Y компоненту (слика 4.2.5). [37]

```
14  Tacka KatmulRom(float t, Tacka p1, Tacka p2, Tacka p3, Tacka p4) {
15      float t2 = t * t;
16      float t3 = t2 * t;
17      Tacka v; // Интерполирана тачкна
18
19      // Израчунавање Катмул-Ром сплајна
20
21      v.x = ((-t3 + 2 * t2 - t)*(p1.x) +
22          (3 * t3 - 5 * t2 + 2)*(p2.x) +
23          (-3 * t3 + 4 * t2 + t)* (p3.x) +
24          (t3 - t2)*(p4.x)) / 2;
25
26      v.y = ((-t3 + 2 * t2 - t)*(p1.y) +
27          (3 * t3 - 5 * t2 + 2)*(p2.y) +
28          (-3 * t3 + 4 * t2 + t)* (p3.y) +
29          (t3 - t2)*(p4.y)) / 2;
30
31      if (!sbIzracunato) std::cout << "v.x = " << v.x << ", " << v.y << std::endl;
32
33      return v;
34  }
```

Слика 4.2.5. - Израчунавање тачака помоћу Катмул-Ром алгоритма

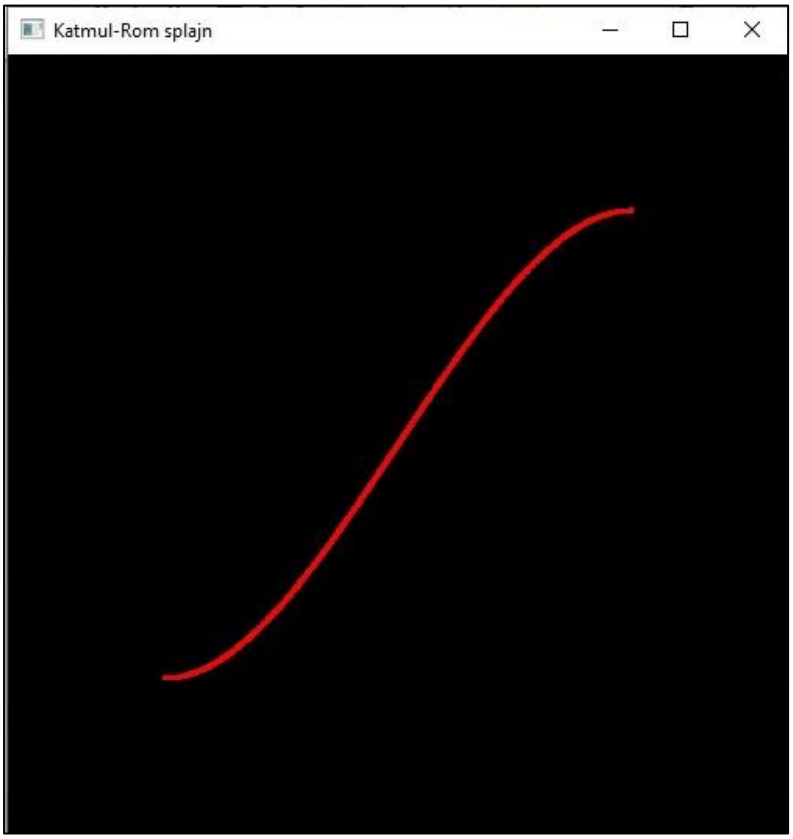
Задајемо четири контролне тачке и започињено интерполацију помоћу Катмул-Ром алгоритма (слика 4.2.6).

```
Tacka p1, p2, p3, p4;

// Наше четири тачке.

p1.x = -90.0f; p1.y = 30.0f;
p2.x = -30.0f; p2.y = -30.0f;
p3.x = 30.0f; p3.y = 30.0f;
p4.x = 90.0f; p4.y = -30.0f;

glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0f, 0.0f, 0.0f);
glPointSize(3); // Величина једне тачке (пиксела).

glBegin(GL_POINTS);
glVertex2f(p1.x, p1.y);
glVertex2f(p2.x, p2.y);
glVertex2f(p3.x, p3.y);
glVertex2f(p4.x, p4.y);
glEnd();

unsigned int brojInterpoliranihTacaka = 0;
for (float t = 0; t < 1; t += 0.001f) {
    brojInterpoliranihTacaka++;
    v = KatmulRom(t, p1, p2, p3, p4);
    glBegin(GL_POINTS);
    glVertex2f(v.x, v.y);
    glEnd();
}
```

Слика 4.2.6. - Унос контролних тачака и интерполација тачака

```
v.x = 28.3795, 29.8711
v.x = 28.4395, 29.8804
v.x = 28.4995, 29.8893
v.x = 28.5595, 29.8979
v.x = 28.6195, 29.9062
v.x = 28.6795, 29.9141
v.x = 28.7395, 29.9217
v.x = 28.7995, 29.9289
v.x = 28.8595, 29.9358
v.x = 28.9195, 29.9423
v.x = 28.9795, 29.9485
v.x = 29.0395, 29.9544
v.x = 29.0995, 29.9599
v.x = 29.1595, 29.965
v.x = 29.2195, 29.9698
v.x = 29.2795, 29.9743
v.x = 29.3395, 29.9783
v.x = 29.3994, 29.9821
v.x = 29.4594, 29.9855
v.x = 29.5195, 29.9885
v.x = 29.5794, 29.9912
v.x = 29.6394, 29.9935
v.x = 29.6994, 29.9955
v.x = 29.7594, 29.9971
v.x = 29.8194, 29.9984
v.x = 29.8794, 29.9993
v.x = 29.9394, 29.9998
v.x = 29.9994, 30
Broj interpoliranih tacaka: 1001
```

Слика 4.2.7. - Приказ координата интерполираних тачака



Слика 4.2.8. - Резултат исцртавања

Координате сваке интерполиране тачке су приказане на слици 4.2.7 док се на слици 4.2.8 види једноставан сплајн исцртан помоћу Катмул-Ром алгоритма. [38]

5. ИНДЕКС ПОЈМОВА

6. ЛИТЕРАТУРА

[1] Б. Строуструп – „The C programming language“ - https://archive.org/details/cprogramminglang00stro_0 – приступано 20.05.2020

[2] Б. Строуструп – „The essence of C++” https://www.youtube.com/watch?v=86xWVb4XIyE – приступано 21.05.2020

[3] Б. Строуструп – Званична веб презентација - http://www.stroustrup.com/ - приступано 22.05.2020.

[4] cplusplus.com – “History of C++” - http://www.cplusplus.com/info/history/ - приступано 23.05.2020.

[5] Д. Ричи – „Development of C programming language” - http://www.bell-labs.com/usr/dmr/www/chist.html – приступано 24.05.2020.

[6] Д. Ричи – „BCPL to B to C“ - https://www.lysator.liu.se/c/dmr-on-histories.html – приступано 25.05.2020.

[7] Д. Меклроj – „Annotated Excerpts from the Programmer’s Manual“ - http://www.cs.dartmouth.edu/~doug/reader.pdf – приступано 26.05.2020.

[8] Б. Строуструп – „Evolving a language in and for the real world“ - http://stroustrup.com/hopl-almost-final.pdf – приступано 27.05.2020.

[9] International Organization for Standardization – „ISO/IEC TS 21544:2018“ - https://www.iso.org/standard/71051.html – приступано 28.05.2020.

[10] C++ Core Guidelines - https://isocpp.github.io/CppCoreGuidelines/ - приступано 29.05.2020.

[11] J. Каплан – „Critique of C++” - http://bearcave.com/misl/misl_tech/c++_critique.html – приступано 06.06.2020.

[12] J. Понтин – „The problem with programming” - https://www.technologyreview.com/2006/11/28/227399/the-problem-with-programming/ - приступано 10.06.2020.

[13] Tutorialspoint / “Computer graphics basics” - https://www.tutorialspoint.com/computer_graphics/computer_graphics_basics.htm – приступано 15.06.2020.

[14] B. Шекат – „Basics of computer graphics“ - http://www.darshan.ac.in/Upload/DIET/Documents/CE/2160703_CG_GTU_Study_Material_2017_11042017_033102AM.pdf – приступано 20.06.2020.

[15] Универзитет Лидс – „Overview of computer graphics” - http://iss.leeds.ac.uk/info/306/giraphics/215/overview_of_computer_graphics/ - приступано 25.06.2020.

[16] Универзитет Јорк – „Milestones in the History of Data Visualization“ - http://www.math.yorku.ca/SCS/Gallery/milestone/milestone.pdf - приступано 29.06.2020.

[17] Toolboom – „Oscilloscopes: History and Classification“ - https://toolboom.com/en/articles-and-video/oscilloscopes-history-and-classification/ - приступано 10.07.2020.

[18] MIT Science Reporter – „Automatically Programmed Tools“ - https://www.youtube.com/watch?v=ob9NV8mmm20 – приступано 13.07.2020.

[19] В. Карлсон – „CG History“ - https://u.osu.edu/waynecarlson/cg-history/ - приступано 15.07.2020.

[20] Универзитет Кембриџ – „EDSAC 1 and after“ - https://www.cl.cam.ac.uk/events/EDSAC99/reminiscences/#EDSAC%201%20people – приступано 20.07.2020.

[21] К. Вудфорд – „Computer graphics“ - https://www.explainthatstuff.com/computer-graphics.html - приступано 24.07.2020.

[22] Microsoft Docs – „Types of bitmaps“ - https://docs.microsoft.com/en-us/dotnet/desktop/winforms/advanced/types-of-bitmaps - приступано 29.07.2020.

[23] И. Гринберг – „Processing: Creative Coding and Computational Art“ - https://books.google.rs/books?id=WTl_7H5HUZAC&pg=PA115 – приступано 03.08.2020.

[24] Р. Граф – „Modern dictionary of electronics“ - http://160592857366.free.fr/joe/ebooks/Electronics%20and%20Electrical%20Engineering%20Collection/GRAF,%20R.%20F.%20(1999).%20Modern%20Dictionary%20of%20Electronics%20(7th%20ed.)/Modern_Dictionary_of_Electronics_7E.pdf - приступано 08.08.2020.

[25] Tutorialspoint – „Computer graphics fractals“ - https://www.tutorialspoint.com/computer_graphics/computer_graphics_fractals.htm - приступано 15.08.2020.

[26] Paroj – „Intro – What is OpenGL?“ - https://paroj.github.io/gltut/Basics/Intro%20What%20is%20OpenGL.html - приступано 25.08.2020.

[27] J. Черников – „Using Modern OpenGL in C++“ - https://www.youtube.com/watch?v=H2E3yO0J7TM – приступано 10.09.2020.

[28] J. Черников – „Vertex Buffers and Drawing a Triangle in OpenGL“ - https://www.youtube.com/watch?v=0p9VxImr7Y0 – приступано 11.09.2020.

[29] J. Черников – „Vertex Attributes and Layouts in OpenGL“ - https://www.youtube.com/watch?v=x0H--CL2tUI – приступано 12.09.2020.

[30] J. Черников – „Writing a Shader in OpenGL“ - https://www.youtube.com/watch?v=71BLZwRGUJE – приступано 13.09.2020.

[31] J. Черников – „Index Buffers in OpenGL“ - https://www.youtube.com/watch?v=MXNMC1YAxVQ – приступано 14.09.2020.

[32] Khronos – „OpenGL Error“ - https://www.khronos.org/opengl/wiki/OpenGL_Error - приступано 15.09.2020.

[33] J. Черников – „Dealing with Errors in OpenGL“ - https://www.youtube.com/watch?v=FBbPWSOQ0-w – приступано 16.09.2020.

[34] Еуклид – „Euclid’s elements“ - https://books.google.rs/books?id=UhgPAAAAIAAJ&redir_esc=y – приступано 17.09.2020.

[35] Б. Станојевић – „Парабола“ - https://matematiranje.in.rs/III%20godina/5.Analiticka%20geometrija/5.PARABOLA.pdf - приступано 18.09.2020.

[36] Г. Фарин, Џ. Хошчек – „Handbook of computer aided geometric design“ - https://books.google.rs/books?id=0SV5G8fgxLoC&pg=PA4 – приступано 19.09.2020.

[37] Javidx9 – „Programming & Using Splines“ - https://www.youtube.com/watch?v=9_aJGUTePYo – приступано 20.09.2020.

[38] Irisraj – “catmull rom spline” - https://community.khronos.org/t/catmull-rom-spline/49524 - приступано 24.09.2020.

7. ПРИЛОЗИ

8. ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

ИЗЈАВА О АКАДЕМСКОЈ ЧЕСТИТОСТИ

Студент (име, име једног родитеља и презиме): **Немања Саша Кузмић**

Број индекса: **НРТ-44/16**

Под пуном моралном, материјалном, дисциплинском и кривичном одговорношћу изјављујем да је завршни рад, под насловом:

C++ апликација за генерисање сплајнова

1. резултат сопственог истраживачког рада;
2. да овај рад, ни у целини, нити у деловима, нисам пријављиво/ла на другим високошколским установама;
3. да нисам повредио/ла ауторска права, нити злоупотребио/ла интелектуалну својину других лица;
4. да сам рад и мишљења других аутора које сам користио/ла у овом раду назначио/ла или цитирао/ла у складу са Упутством;
5. да су сви радови и мишљења других аутора наведени у списку литературе/референци који је саставни део овог рада, пописани у складу са Упутством;
6. да сам свестан/свесна да је плагијат коришћење туђих радова у било ком облику (као цитата, прафраза, слика, табела, дијаграма, дизајна, планова, фотографија, филма, музике, формула, вебсајтова, компјутерских програма и сл.) без навођења аутора или представљање туђих ауторских дела као мојих, кажњиво по закону (Закон о ауторском и сродним правима), као и других закона и одговарајућих аката Високе школе електротехнике и рачунарства струковних студија у Београду;
7. да је електронска верзија овог рада идентична штампаном примерку овог рада и да пристајем на његово објављивање под условима прописаним актима Високе школе електротехнике и рачунарства струковних студија у Београду;
8. да сам свестан/свесна последица уколико се докаже да је овај рад плагијат.

У Београду, __. 10. 2020. године

Својеручни потпис студента