SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zavod za elektroniku, mikroelektroniku,
računalne i inteligentne sustave

RAČUNALNA GRAFIKA

Laboratorijske vježbe

Željka Mihajlović, Karla Brkić, Marko Čupić

Zagreb, 2016.

UVOD

Laboratorijske vježbe sastavni su dio izučavanja gradiva kolegija računalne grafike. Praktičnim radom u laboratoriju usvaja se, utvrđuje i proširuje znanje iz računalne grafike.

Težnja je da se laboratorijske vježbe realiziraju upotrebom verzije OpenGL 3.x, odnosno naprednijih verzija OpenGLa koji koriste sjenčare. Kako bi se olakšali prvi koraci u uputama se nalazi pojašnjenje kako podesiti postavke MS VisualStudia u Dodatku A, a Dodatak B sadrži razradu nekoliko primjera korištenjem sjenčara. Potrebno je proći kroz ove primjere prije početka implementacije vježbi koje je potrebno ostvariti. U prvoj vježbi potrebno je ostvariti praćenje putanje određene aproksimacijskom uniformnom B-splajn krivuljom uz podudaranje orijentacije s tangentom krivulje. U drugoj vježbi potrebno je napraviti vlastiti pokretač sustava čestica.

1. Praćenje putanje

Za putanju po kojoj ćemo pomicati naš objekt odabrat ćemo aproksimacijsku uniformnu B-splajn kubnu krivulju zbog njenog jednostavnog oblika, različitih poželjnih svojstava, te jednostavnog načina određivanja derivacije. Derivacija određuje tangentu na krivulju, a koristit ćemo ju za određivanje orijentacije objekta.

1.1 Aproksimacijska uniformna B-splajn kubna krivulja

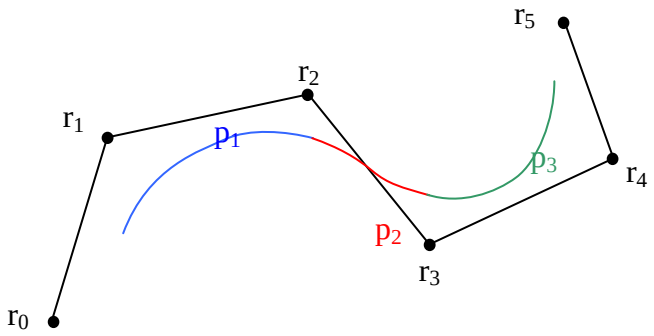
Aproksimacijska B-splajn krivulja definirana je:

$$\vec{p}(u) = \sum_{i=0}^{n} \vec{r}_i \, N_{i,3}(u) \tag{1.1}$$

Za aproksimacijsku B-splajn kubnu krivulju možemo imamo uniformno zadan vektor uzlova u_{i+1}-u_i = *konstanta*. To znači da su uzlovi jednoliko raspoređeni. Ako je razmak jedinični vrijedi u_{i+1}-u_i = 1. U tom slučaju krivulju možemo načiniti upotrebom samo periodičkog segmenta. Periodički segment aproksimacijske kubne B-splajn krivulje određen je sa četiri točke, a u tom slučaju *i*-ti segment krivulje zadan je:

$$\vec{p}_i(t) = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \vec{r}_{i-1} \\ \vec{r}_i \\ \vec{r}_{i+1} \\ \vec{r}_{i+2} \end{bmatrix} = \mathbf{T}_3 \mathbf{B}_{i,3} \mathbf{R}_i , \tag{1.2}$$

gdje se za svaki segment parametar t mijenja $0 \le t < 1$. Sa četiri točke određen je jedan segment krivulje, a općenito s n točaka određeno je n-3 segmenata krivulje (Slika 1).



Slika 1. Početni segment p_1 (plavi) krivulje određen je točkama r_0, r_1, r_2, r_3, slijedeći segment p_2 (crveni) određen je točkama r_1, r_2, r_3, r_4, a završi segment p_3 (zeleni) određen je točkama r_2, r_3, r_4, r_5.

1.2 Tangenta na aproksimacijsku uniformnu B-splajn kubnu krivulju

Tangenta na krivulju zadanu jednadžbom (1.2) određena je parametarskom derivacijom po parametru *t*. Tangenta $\vec{p}_i'(t)$ u točki krivulje određenoj parametrom *t* određena je istim kontrolnim točkama i matričnim oblikom:

$$\frac{\partial \vec{p}_i(t)}{\partial t} = \vec{p}_i'(t) = \begin{bmatrix} 3t^2 & 2t & 1 & 0 \end{bmatrix} \mathbf{B}_{i,3}\, \mathbf{R}_i \tag{1.3}$$

odnosno, jednadžbu (1.3) možemo zapisati:

$$\vec{p}_i'(t) = \begin{bmatrix} t^2 & t & 1 \end{bmatrix} \frac{1}{2} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 2 & -4 & 2 & 0 \\ -1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \vec{r}_{i-1} \\ \vec{r}_i \\ \vec{r}_{i+1} \\ \vec{r}_{i+2} \end{bmatrix} = \mathbf{T}_2\, \mathbf{B}'_{i,3}\, \mathbf{R}_i \tag{1.4}$$

Treba primijetiti da izraz (1.2) određuje točku krivulje $\vec{p}_i(t)$, a izraz (1.3) odnosno (1.4) smjer tangente $\vec{p}_i'(t)$, odnosno **vektor** u toj točki.

1.3 Orijentacija objekta

Položaj objekta u prostoru određen je pozicijom objekta u radnom prostoru koju određuju tri prostorne koordinate $\vec{p}_i(t) = [x(t) \quad y(t) \quad z(t)]$, te orijentacijom objekta koja je određena vektorom orijentacije. Objekt je potrebno rotirati oko koordinatnih osi kao bi postigao položaj zadane orijentacije, no nije svejedno kojim redoslijedom i oko kojih osi (izvornih ili rotiranih) ćemo obavljati rotaciju. Orijentacija se može zapisati i preko kutova obzirom na koordinatni sustav scene ili koordinatni sustav objekta, te definiranog redoslijeda koordinatnih osi oko kojih se obavlja rotacija. Ovi kutovi zovu se Eulerovi kutovi. U koordinatnom sustavu objekta promjene kutova (Slika 2.) određuju se po nagibu/dubini/smjeru (engl. roll/pitch/yaw).

Slika 2. Nagib dubina i smjer u koordinatnom sustavu objekta.

Promjena orijentacije zahtijeva interpolaciju u nizu zadanih orijentacijskih položaja. U postupku promjene orijentacijskog položaja kako u mehaničkim tako i u numeričkim postupcima može doći do problema kod podudaranja koordinatnih osi. Taj problem se naziva zastoj žiroskopskih osi (engl. gimbal lock). Problem se rješava tako da se dodaje još jedan stupanj slobode. Mehanički se to izvodi dodavanjem još

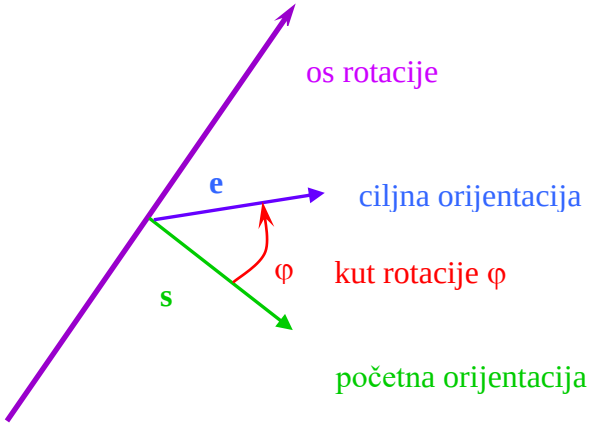
jednog prstena na žiroskop. U OpenGL-u rješenje je ostvareno uvođenjem još jedne varijable. Upotrebom kvaterniona za zapis i interpolaciju orijentacije objekta postiže se kvalitetno rješenje.

1.4 Rotacija objekta u OpenGL-u

Rotacija objekta u OpenGL zadaje se upotrebom četiri parametra. To su os rotacije (tri parametra) i kut rotacije (Slika 3).

Početna i ciljna orijentacija objekta svojim vektorskim produktom određuju os rotacije. Početna orijentacija objekta neka je zadana vektorom **s**, a orijentacija koju želimo postići vektorom **e**. Početnu orijentaciju možemo proizvoljno odabrati. Ako je nos aviona orijentiran u smjeru z-osi odabrat ćemo za vektor **s**=(0 0 1). Vektor **os** određuje os oko koje je potrebno rotirati objekt.

$$\mathbf{os} = \mathbf{s} \times \mathbf{e} = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ s_x & s_y & s_z \\ e_x & e_y & e_z \end{vmatrix} = \begin{bmatrix} \vec{i} & \vec{j} & \vec{k} \end{bmatrix} \cdot \begin{bmatrix} s_y e_z - e_y s_z \\ -(s_x e_z - e_x s_z) \\ s_x e_y - s_y e_x \end{bmatrix} \tag{1.5}$$

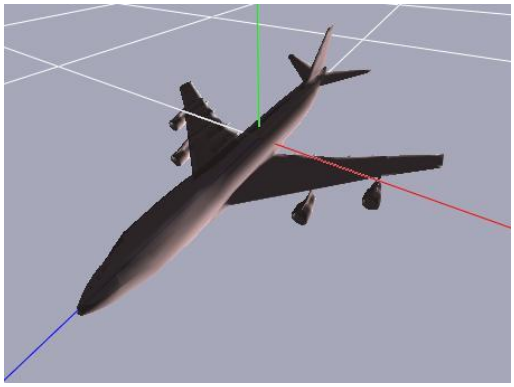


Slika 3. Os rotacije i kut rotacije.

Kosinus kuta rotacije određen je skalarnim produktom vektora početne i ciljne orijentacije objekta podijeljenim s normama tih vektora:

$$\cos\varphi = \frac{\mathbf{s}\,\mathbf{e}}{|s||e|} \tag{1.6}$$

U naredbi OpenGL-a glRotatef(φ, os_x, os_y, os_z) prvi parametar određuje kut rotacije izražen u stupnjevima gledano iz pozitivnog smjera osi suprotno od kazaljke na satu, a slijedeća tri parametra određuju pojedine komponente vektora osi rotacije. Obratiti požnju da funkcija acos() obično vraća kut u radijanima a glRotate() prihvaća kut u stupnjevima. Početna orijentacija **s** objekta može se odabrati proizvoljno, a odredimo ju tako da početna orijentacija objekta bude u smjeru u kojem želimo da se objekt giba (Slika 4).



Slika 4. Za početni položaj objekta u koordinatnom sustavu *x*-crvena, *y*-zelena, *z*-plava, početnu orijentaciju **s** objekta odabrat ćemo tako da se podudara sa *z* osi.

Ako početna orijentacija (pozicija) objekta nije onakva kavu bi željeli, potrebno je inicijalno rotirati objekt (trenslatirati) kako bi se središte objekta i početna orijentacija podudarila s putanjom.

1.5 Akumulacija pogreške

Prilikom transformacije objekta mijenjaju se izračunate koordinate objekta. Računanje novih koordinata na osnovi starih nije poželjno zbog nakupljanja numeričke pogreške. Izvorne koordinate objekta u lokalnom koordinatnom sustavu objekta se čuvaju, a novi položaj objekta određuje se iz izvornih koordinata objekta i zadanog ciljnog položaja koji objekt treba zauzeti. To znači da ćemo za svaki izračunati položaj objekta i orijentaciju objekta množiti izvorne (početne) koordinate objekta matricom rotacije i translacije uz odgovarajući redoslijed, a zatim ćemo iscrtati objekt.

1.6 Reprezentacije orijentacije DCM matricom

Orijentaciju objekta možemo postići ako znamo poziciju i koordinatne osi ciljnog koordinatnog sustava. U našem primjeru pomak na zadano mjesto krivulje određuje poziciju. Koncentrirat ćemo se sada samo na orijentaciju, tako da ćemo promatrati poziciju fiksiranu uz ishodište koordinatnog sustava O(0, 0, 0).

Od koordinatnog sustava koji određuje orijentaciju našeg objekta znamo jednu os koja je određena tangentom (1.4). Potrebno je odrediti i druge dvije koordinatne osi, odnosno normalu i binormalu kao bi imali potpuno određen koordinatni sustav objekta. Normala je određena vektorskim umnoškom prve i druge derivacije, odnosno vektorom tangente kojeg imamo (1.4) i drugom derivacijom koji dobijemo ako (1.4) još jednom deriviramo po parametru *t*:

$$\vec{u} = \vec{p}_i'(t) \times \vec{p}_i''(t) \tag{1.7}$$

a binormala je određena vektorom tangente $\vec{w} = \vec{p}_i'(t)$ i dobivenim vektorom normale $\vec{u}$, te tako ovi vektori formiraju desni koordinatni sustav:

$$\vec{v} = \vec{w} \times \vec{u} \tag{1.8}$$

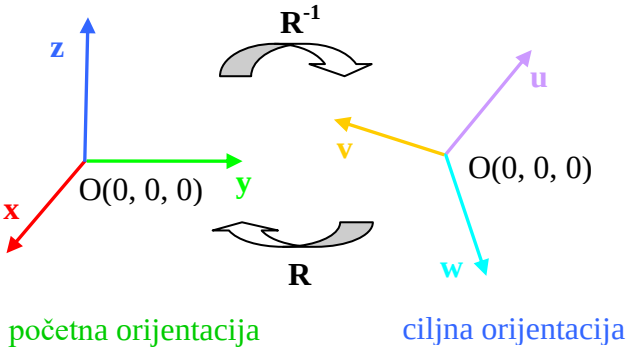
Treba pripaziti na drugu derivaciju koja može biti nula u slučaju kada je segment krivulje ravan (zadane četiri točke su na pravcu). U tom slučaju obično se odabire neka orijentacija.

Određivanjem vektora tangenta, normale i binormale dobili smo desni koordinatni sustav određen koordinatnim osima $\vec{w}, \vec{u}, \vec{v}$. Ako je zadan objekt u koordinatno sustavu $\vec{w}, \vec{u}, \vec{v}$ a želimo ga transformirati u $\vec{x}, \vec{y}, \vec{z}$ tako da se navedenim redom podudare koordinatne osi, potrebno je pomnožiti sve točke zadane u sustavu $\vec{w}, \vec{u}, \vec{v}$ rotacijskom matricom:

$$\mathbf{R} = \begin{bmatrix} \vec{w} & \vec{u} & \vec{v} \end{bmatrix} = \begin{bmatrix} w_x & u_x & v_x \\ w_y & u_y & v_y \\ w_z & u_z & v_z \end{bmatrix} \tag{1.9}$$

kako bi dobili koordinate u sustavu $\vec{x}, \vec{y}, \vec{z}$ (Slika 5). Ova matrica se zove rotacijska ili DCM matrica (*direction cosine matrix*). Neka svojstva ove matrice su da ima realne parametre, ortogonalna je, determinanta joj je 1, svojstveni vektor čija je pripadna svojstvena vrijednost 1 određuje os rotacije.

Nama je potrebna obrnuta transformacija, odnosno objekt je zadan u $\vec{x}, \vec{y}, \vec{z}$ a potrebno je transformirati objekt u $\vec{w}, \vec{u}, \vec{v}$, odnosno potrebno je pomnožiti točke objekta iz lokalnog koordinatnog sustava objekta $\vec{x}, \vec{y}, \vec{z}$ inverznom matricom **R** (1.9) kako bi dobili koordinate u sustavu $\vec{w}, \vec{u}, \vec{v}$.



Slika 5. Promjena orijentacije *x, y, z* u *w, u, v*.

Ovaj način određivanja orijentacije treba povezati još s pomakom, odnosno prvo odredimo orijentaciju, a zatim pomaknemo objekt na traženu poziciju u prostoru. Ovisno o inicijalnoj orijentaciji objekta u ovoj vježbi, tj. je li trup aviona (Slika 4) u smjeru *x, y, z* osi bit će potrebno promijeniti redoslijed podudaranja osi ili inicijalno rotirati objekt tako da se tangenta krivulje podudari sa željenom orijentacijom pri kretanju po krivulji.

1.7 Radni zadatak

Zadan je jednostavan primjer u kojem se koristi samo dio zapisa objekta Wavefront (.obj). Zapis sadrži popis vrhova i njihovih koordinata te popis poligona s pripadnim indeksima vrhova.

1. Iz datoteke učitati zadano tijelo (dio laboratorijske vježbe 4. iz Interaktivne računalne grafike).
2. Iz datoteke učitati niz točaka koje određuju aproksimacijsku uniformnu kubnu B-splajn krivulju.
3. Za svaki segment krivulje mijenjati parametar *t* od 0 do 1:
   - 3.1. Prema formuli (1.3) ili (1.4) odrediti ciljnu orijentaciju objekta. Na osnovi početne orijentacije i ciljne orijentacije odrediti os rotacije i kut rotacije po formulama (1.5) i (1.6). Prema formuli (1.2) odrediti potrebnu translaciju objekta.
   - 3.2. Odrediti transformirane koordinate objekta.
   - 3.3. Iscrtati krivulju (putanju) i pripadnu tangentu. Tangente nacrtati tako da se od pojedine točke krivulje nacrta kratka dužina u smjeru tangente (ciljne orijentacije) za parametar *t*. Znači početna točka dužine je točka krivulje, a završna se dobije tako da se na početnu točku zbroji vektor tangente skaliran s proizvoljnim faktorom.
   - 3.4. Iscrtati objekt.
   - 3.5. Usporediti načine određivanja orijentacije opisane u poglavlju 1.4 i 1.6.
4. Definirati putanju u obliku spirale npr.

V_1=(0 0 0) V_2=(0 10 5) V_3=(10 10 10) V_4=(10 0 15)

V_5=(0 0 20) V_6=(0 10 25) V_7=(10 10 30) V_8=(10 0 35)

V_9=(0 0 40) V_{10}=(0 10 45) V_{11}=(10 10 50) V_{12}=(10 0 55)

animirati objekt.

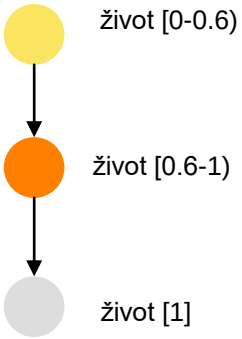
2. Sustav čestica

Za opis nekih objekata i pojava, objekti zadani nizom poligona nisu pogodni. To su obično neizraziti objekti (pojave) kao na primjer vodoskok, vatra, snijeg, kiša, zviježđe. Za ovakve objekte pogodan način opisa je sustav čestica čija nakupina predstavlja objekt. Svojstvo takvog objekta je nedeterminističko ponašanje i potrebno je koristiti stohastičke procese pri kreiranju i ponašanju sustava. Rad sa česticama obično nije vremenski zahtjevan (ovisi o broju čestica), pa je moguće ostvariti vrlo složena dinamička svojstva objekta i vizualne učinke. Pored zanimljivih vizualnih učinaka sustavi čestica se koriste u simulacijama raznih fizikalnih procesa.

2.1 Definiranje čestice

Sustav čestica možemo definirati kao objekt u svom koordinatnom sustavu, s pripadnim nizom atributa zajedničkim za sustav kao, odnosno individualnim za pojedinu česticu. Za pojedinu česticu važno je definirati njene glavne značajke vezane uz pojavu čestice, odnosno za pojedine faze potrebno je odrediti:

- **rađanje čestice** – u trenutku stvaranja čestice potrebno je definirati početne uvijete
  - trenutak (učestalost) stvaranja čestica,
  - mjesto nastajanja. Kod vodoskoka će to biti jedan izvor čestica, a kod snijega je to obično definirano poligonom koji emitira niz čestica.
  - početni uvjeti, kao što je smjer gibanja i početna brzina.
- **život čestice** – potrebno je odrediti trajanje (brzina umiranja može biti različita), odnosno životni vijek i atribute koji opisuju promjene za vrijeme života jedne čestice. To su na primjer:
  - putanja čestice odnosno promjenu pozicije u vremenu. Promjena pozicije se obično opisuje jednostavnim ili složenijim fizikalnim zakonima npr. $\Delta s = v\,\Delta t$, u obzir možemo uzeti i utjecaj početne brzine, djelovanje sile (gravitacije) na česticu ili proizvoljne sile koje definiramo i sl.
  - promjena boje (oblika), prozirnosti, Slika 1. prikazuje primjer promjene boje tijekom životnog vijeka čestice.
  - način interakcije s nekim drugim objektom, na primjer kolizija i odbijanje od podloge ili međudjelovanje samih čestica. Ako je broj čestica velik međudjelovanje čestica se obično izbjegava zbog vremenske zahtjevnosti.
- **umiranje čestice** – memorijski prostor koji je bio korišten za pojedinu česticu obično se koristi za stvaranje i zapis nove čestice. Alokacija memorije (i oslobađanje) nepotrebno može trošiti vrijeme ako ju radimo pri svakom rađanju (umiranju) čestica. Upotrebom zadanog memorijskog bloka možemo postići prisutnost podjednakog broja čestica u vremenu. Nedostatak unaprijed zadanog memorijskog bloka je potrebna procjena maksimalnog broja čestica i neefikasno korištenje memorije ako se ta veličina mijenja.



Slika 1. Promjena boje tijekom životnog vijeka čestice, kod simulacije vatre.

Prilikom definiranja pojedinih parametara obično se dodaje utjecaj slučajne veličine, a distribucija će nam odrediti, na primjer, širinu mlaza vodoskoka.

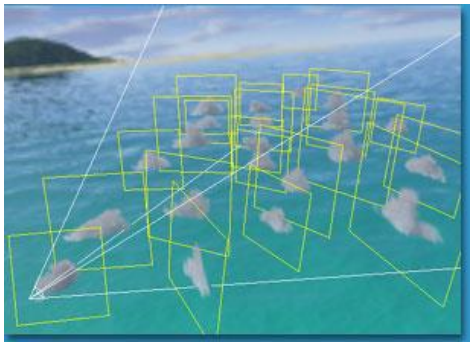
Pojedini atributi vezani su uz samu česticu, dok su neki zajednički za cijeli sustav. Tako, na primjer ako imamo sustav u kojem sve čestice imaju jednako vrijeme života, taj atribut možemo izbaciti iz strukture pojedine čestice i definirati ga kao zajednički. Veliki broj atributa možemo proglasiti individualnim, no to vodi do povećanja računalnih zahtjeva za prikaz sustava, pa treba naći kompromis između složenosti ponašanja pojedine čestice i broja čestica koji se koristi.

2.2 Prikaz čestica

Pojedinu česticu možemo prikazati točkom, uz povećanu dimenziju kako bi bila bolje vidljiva u konačnom prikazu. Za složeniji oblik čestice, pojedinu česticu nije prikladno prikazivati mrežom poligona zbog vremenske zahtjevnosti. Prihvatljiv kompromis obično je prikaz čestice malim poligonom s teksturom (Slika 2.) čime se povećava složenost prikazanog sustava. Pripadna tekstura može predstavljati česticu, pahuljicu snijega, a rubovi su obično prozirni (RGB komponente određuju boju, a α vrijednost određuje prozirnost). Format zapisa koji podržava prozirnost je npr. tga. Možemo koristiti i okolnu boju prilagođenu boji pozadine (Slika2. cestica.bmp).

Slika 2. Primjer teksture za prikaz čestice: iskrica.tga, cestica.bmp, explosion.bmp, snow.bmp, smoke.bmp, Bee.jpg.

Kako je poligon dvodimenzionalan nećemo dobiti dobar učinak ako ga promatramo ”iz profila”, pri promjeni očišta, pa je potrebno poligon orijentirati u prostoru tako da normala bude okrenuta uvijek prema promatraču. Takav način orijentacije poligona prikazan je na Slici 3, a naziva se tehnika oglasne ploče (engl. billboard). Promjena orijentacije poligona obavlja se način koji je opisan u prethodnoj vježbi (odredi se os oko koje se rotira poligon i kut za koji se rotira na osnovi početne orijentacije zadanog poligona s teksturom i vektora prema očištu za zadani položaj na koji treba postaviti česticu).



Slika 3. Tehnika oglasne ploče, pojedini poligoni s pripadnom teksturom okrenuti su prema promatraču.

2.2.1 Učitavanje teksture

Učitavanje teksture olakšano je korištenje biblioteka. Na primjer biblioteka `glaux.h` podržava učitavanje slika u .bmp formatu. Bolja i općenitija podrška za učitavanje i spremanje slika je DevIL (http://openil.sourceforge.net/).

2.3 Čestični pogon

Proces proračunavanja novog stanja sustava i prikaz na zaslon sastoji se od nekoliko koraka. Prilikom proračuna novog stanja pratimo vrijeme koje proteklo od prošle promjene. Na početku proračuna svim se česticama povećava starost za proteklo vrijeme. Odrede se čestice čija je starost veća od njihovog vremena života i te čestice se gase. Pokreće se proces stvaranja novih čestica i određuju se pripadne početne vrijednosti atributa tih čestica (pozicija, smjer, veličina brzine, duljina života i sl.). Određuje se novi položaj i drugi parametri ostalih čestica. Transformira se sustav čestica u sustav scene na novi položaj i poligoni s teksturama se okreću prema promatraču, nakon čega slijedi prikaz čestica i proračun novog stanja.

2.4 Dodatne mogućnosti sustava čestica

Pomicanje izvora čestica u prostoru. Sustav čestica definiran kao objekt u svom koordinatnom sustavu možemo pomicati u prostoru po proizvoljnoj krivulji kao točkasti izvor čestica ili možemo pomicati promatrača kroz oblak čestica. U jednoj sceni možemo imati i nekoliko sustava čestica.

Periodičke promjene brojnosti čestica. Promjenom frekvencije stvaranja novih čestica možemo ostvariti periodičke promjene u brojnosti generiranih čestica. Na ovaj način možemo ostvariti slanje ”dimnih signala” sustavom čestica.

Utjecaj sile na čestice sustava. Utjecaj sile na čestice možemo ostvariti tako da dodamo utjecaj gravitacije ili možemo dodati točke privlačenja koje će djelovati na čestice sustava. Utjecaj takvih točaka privlačenja obično opada s udaljenošću ili kvadratom udaljenosti. U ovom slučaju koristiti prikaz čestice točkom umjesto teksturom.

$$F_{ukupno} = \sum F_i = m_{čestice} \cdot a\,,\ a = \frac{F_{ukupno}}{m_{čestice}}$$

$$\dot{v} = a \qquad v_{n+1} = v_n + a \cdot \Delta t$$

$$\dot{x} = v \qquad x_{n+1} = x_n + v \cdot \Delta t$$

Kolizija s dodatnim objektima u sceni. U scenu možemo dodati jednu ili nekoliko ravnina s kojima ćemo ostvariti koliziju, odnosno ispitivati ćemo da li je čestica udarila u promatrane poligone i definirat ćemo da li je došlo do odbijanja čestice ovisno o kutu upada ili do klizanja čestice po promatranoj površini (elastični ili plastični sudar. U ovom slučaju koristiti prikaz čestice točkom umjesto teksturom.

Promjene boje i veličine čestice. Tijekom života čestice možemo mijenjati boju prema Slici 1 (prozirnost) i veličinu čestice i na taj način postići dinamičnost načinjenog sustava.

Različiti oblici izvora čestica. Već je spomenuto da izvor čestica može biti točkasti, poligonalan, no možemo načiniti i da površina cijelog objekta bude mogući izvor čestica. Na ovaj način ostvarit ćemo učinak kao da se objekt dimi.

Modeliranje nakupine objekata sustavom čestica. Sustavi čestica mogu se koristiti i pri modeliranju nakupine objekata. Roj pčela, na primjer, možemo načiniti kao sustav čestica, gdje je kretanje pojedine jedinke opisano kretanjem čestice, a pojedina jedinka ovisno o blizini promatrača može se prikazati kao tekstura ili trodimenzijski model.

Sklopovski podržana implementacija sustava čestica. Ako koristimo sklopovsku opremu (GPU) za zapis sustava čestica, tada je pogodno koristiti teksturu za zapis atributa pojedine čestice. Na primjer, vektor brzine (*xyz* komponente) možemo čuvati u RGB komponentama teksture. Procesorske jedinice tada možemo upotrijebiti za proračun novog položaja.

Sustav čestica sa sustavom opruga. Općenito, neki objekt možemo učitati kao strukturu u kojoj vrhovima pridijelimo svojstva čestice (mase), a bridovima pridijelimo svojstva opruge. Na ovaj način ostvaruje se model pogodan za opis tkanine, no i bilo koji drugi objekt (teddy) poprima elastična svojstva.

2.5 Radni zadatak

Potrebno je načiniti sustav čestica odnosno čestični pogon kako je opisano u podpoglavlju 2.3. Odabrati jednu od dodatnih mogućnosti sustava čestica predloženih u podpoglavlju 2.4 i ugraditi u svoju implementaciju sastava čestica ili umjesto predloženih dodatnih mogućnosti sustava čestica realizirati vlastiti prijedlog dodatne mogućnosti.

DODATAK A

Podešavanje integrirane razvojne okoline Microsoft Visual Studio za uporabu OpenGL-a inačice 3.x

Ove upute će vam pomoći da ispravno podesite razvojnu radnu okolinu Microsoft Visual Studio kako biste mogli pokretati primjere pisane u OpenGL-u verzije 3.3 koji su dostupni na stranicama predmeta. Upute su napisane za Microsoft Visual Studio 2013, ali isti principi vrijede i za druge verzije.

Prvi korak je preuzimanje zip arhive koju smo vam pripremili na stranici kolegija. Raspakirana arhiva ima sljedeću strukturu:

```
\IRG
      \Biblioteke
            freeglut-2.8.1
            glew-1.13.0
            glm
      \Primjeri
            primjer01
            primjer02
            primjer02b
            primjer02c
            primjer03
            primjer04
            primjer05
```

U ovim uputama pretpostavljat ćemo da ste arhivu raspakirali u direktorij Y:\. Prilikom podešavanja u svim putanjama u ovim uputama potrebno je zamijeniti Y:\ vršnim direktorijem u koji ste raspakirali direktorij IRG.

Važna napomena: u ovim uputama pretpostavljamo da putanja do raspakiranog direktorija IRG ne sadrži razmake. Ukoliko u putanji ipak imate razmake, na svim mjestima gdje je u Visual Studio potrebno upisati neku putanju istu okružite dvostrukim navodnicima.

Primjer: ako u uputama piše da treba unijeti putanju Y:\IRG\Biblioteke\glew-1.13.0\lib\Release\Win32

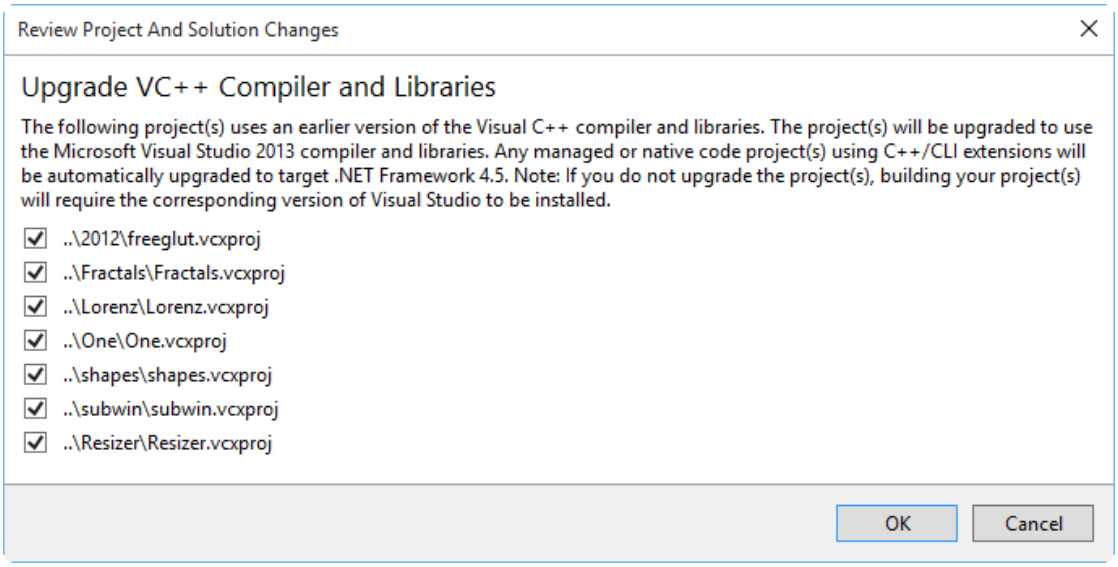
a vaša zamjena za direktorij Y:\, tj. mjesto gdje ste raspakirali arhivu je C:\Moji projekti\, onda umjesto da kao konfiguraciju upišete:

C:\Moji projekti\IRG\Biblioteke\glew-1.13.0\lib\Release\Win32

potrebno je upisati

"C:\Moji projekti\IRG\Biblioteke\glew-1.13.0\lib\Release\Win32"

Nakon što ste raspakirali arhivu, potrebno je najprije izgraditi biblioteku freeglut. Locirajte datoteku Y:\IRG\Biblioteke\freeglut-2.8.1\VisualStudio\2012\freeglut.sln i dvaput kliknite na nju. Otvorit će se Visual Studio, te će se u slučaju da radite s verzijom 2013 pojaviti sljedeći prozor:

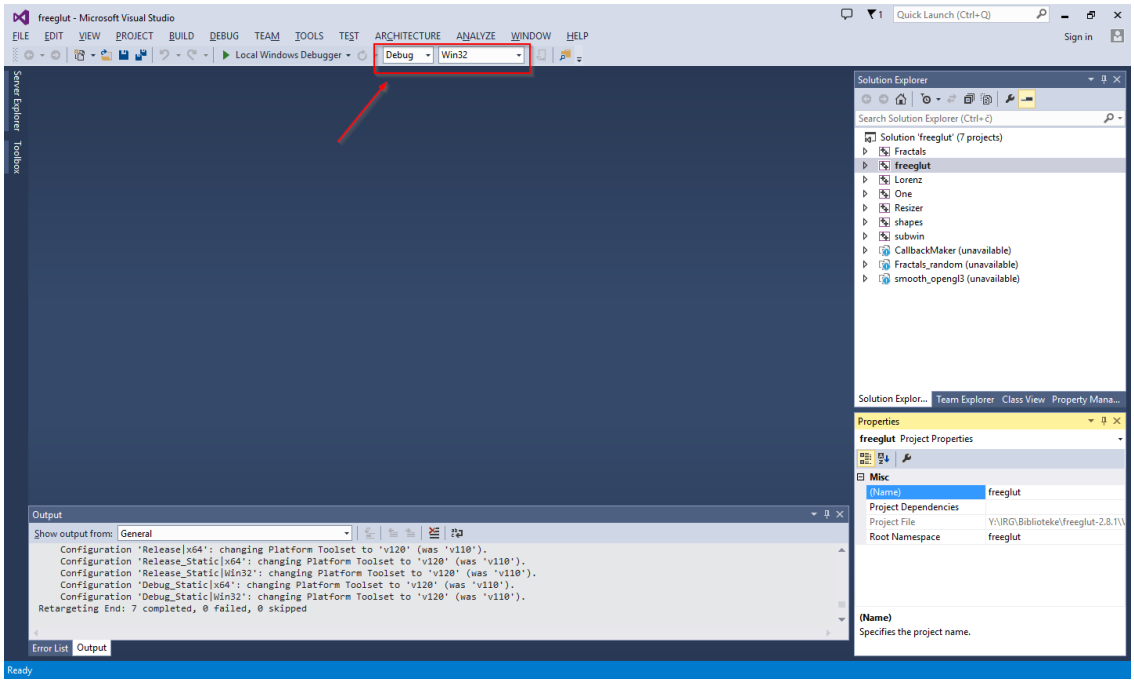


Slika A.1. Izgradnja biblioteke freeglut

Odaberite OK.

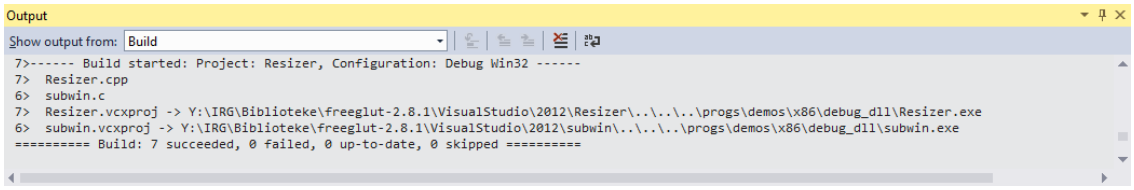
Provjerite da su vam postavke za izgradnju biblioteke podešene kao na donjoj slici (Debug, Win32[1]).

[1] Ukoliko želite, moguće je razvijati i za 64-bitne arhitekture (tada biste ovdje odabrali x64, kao i svugdje kasnije). Međutim, za to nema potrebe u okviru ovih laboratorijskih vježbi pa smo u ovim uputama pretpostavili 32-bitnu arhitekturu.



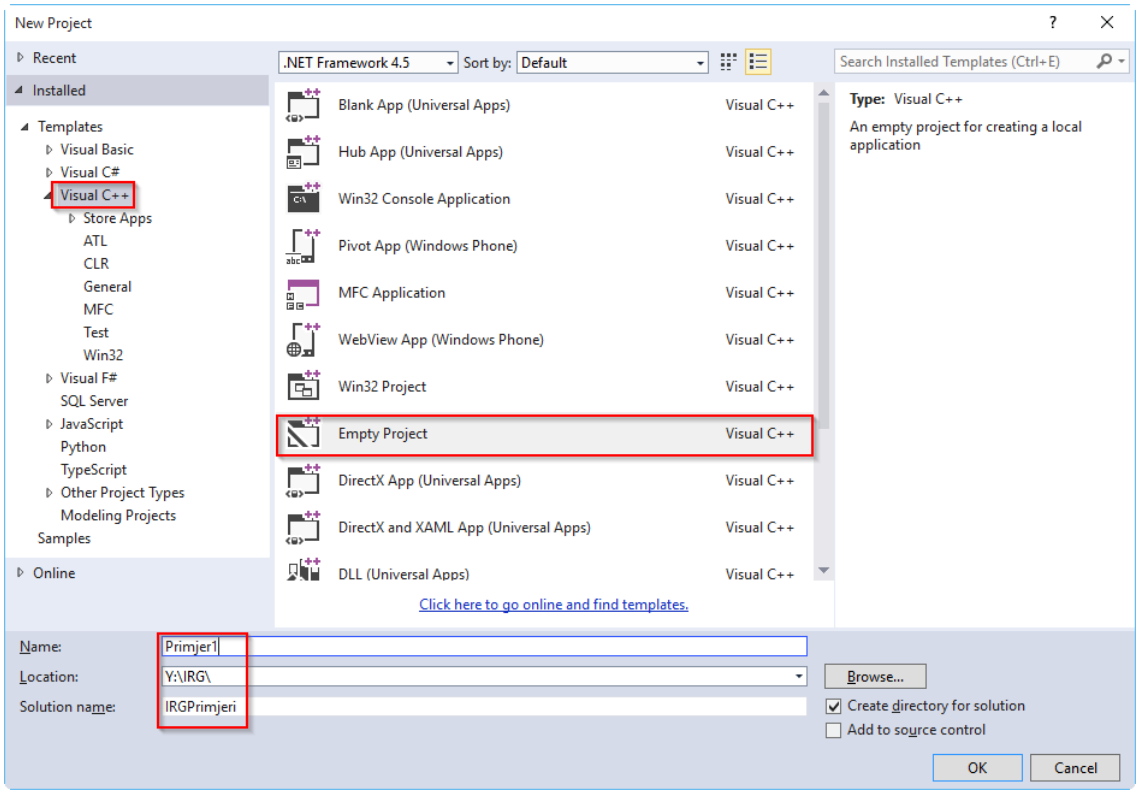
Slika A.2. Postavke za izgradnju biblioteke freeglut

Odaberite Build → Build Solution. Nakon nekog vremena izgradnja će završiti te bi u vašem izlaznom panelu na dnu prozora trebala biti poruka kao na slici dolje:



Slika A.3. Uspješno izgrađena biblioteka freeglut

Time je naš posao s bibliotekom freeglut gotov. Zatvorite solution biblioteke, te krećemo sa stvaranjem vlastitog projekta za laboratorijske vježbe. U Visual Studiju odaberite File → New Project… te podesite parametre kao na slici dolje:

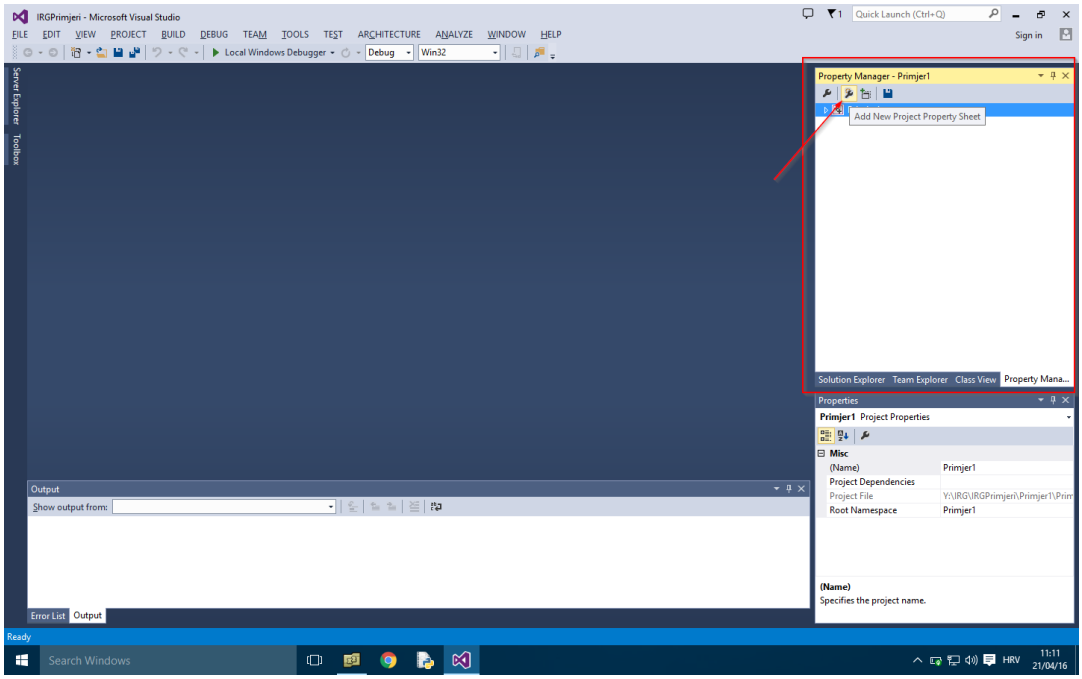


Slika A.4. Postavke parametara vlastitog projekta u Visual Studiju

Odaberite Visual C++, prazan projekt, lokaciju po želji i kliknite OK.

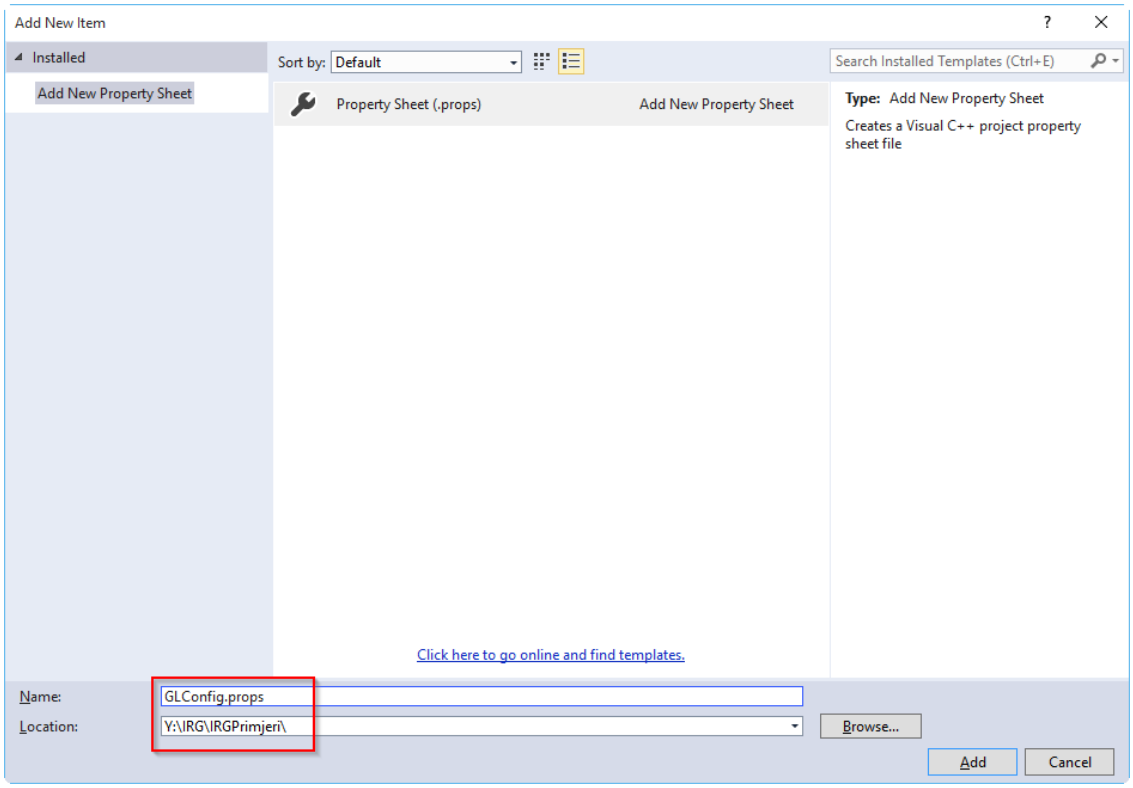
U sljedećem koraku definirat ćemo da naš projekt koristi biblioteke freeglut, GLEW i glm, te ćemo podesiti sve što je potrebno kako bi ga mogli izgraditi. Kako ovo ne bi morali ponavljati za svaki novi projekt s istim ovisnostima koji budemo radili, koristit ćemo tzv. property sheetove. Property sheetovi su ugrađena funkcionalnost Visual Studija koja nam omogućuje da popišemo potrebne include direktorije, biblioteke za linker i sl. i snimimo te informacije u jedinstvenu datoteku (property sheet). Svaki sljedeći put kad budemo željeli imati projekt koji radi s našim bibliotekama bit će dovoljno samo učitati prethodno definiran property sheet.

U glavnom prozoru Visual Studija odaberite View → Other windows → Property Manager. U panelu s desne strane ekrana pojavit će se Property Manager. Odaberite ikonu za dodavanje novog property sheeta kao na slici dolje:



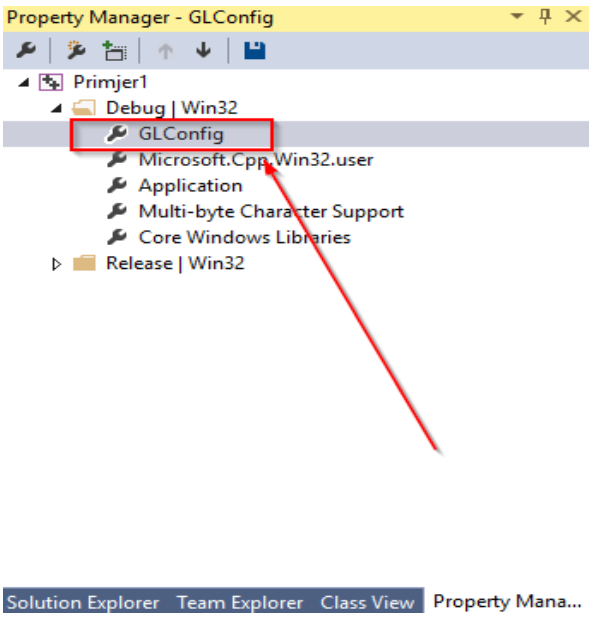
Slika A.5. Property Manager u Visual Studiju

Pojavit će se prozor kao na slici:



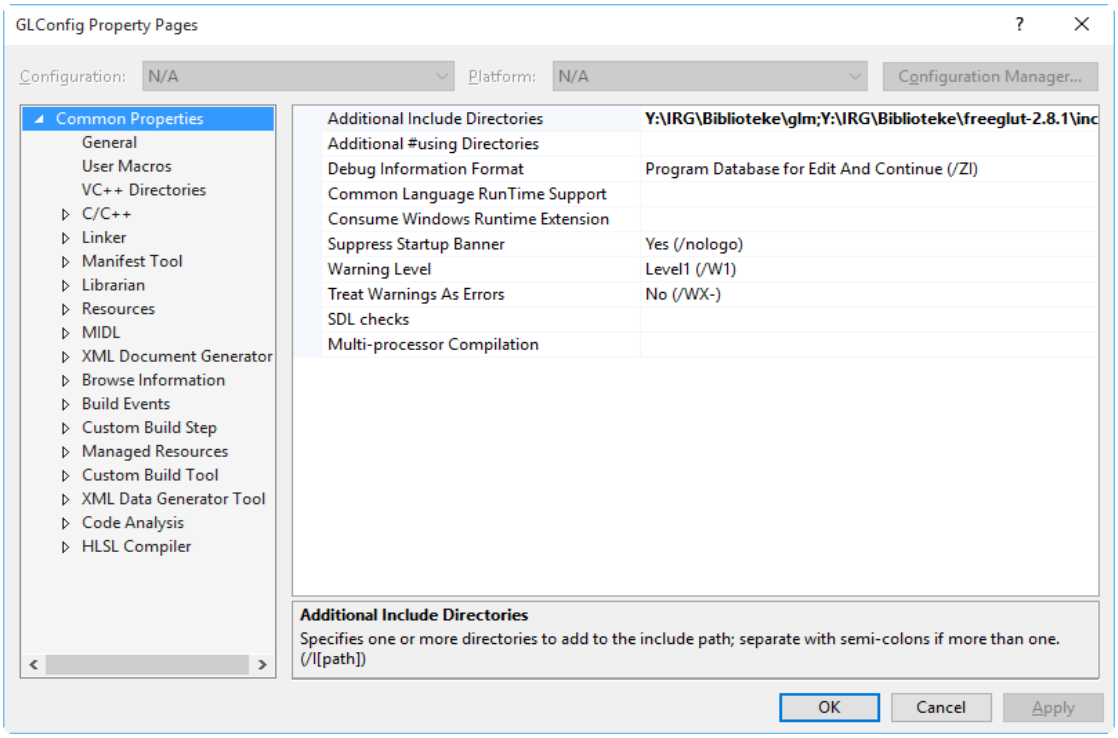
Slika A.6. Property sheet u Visual Studiju

Odaberite ime za property sheet (u našem primjeru to je GLConfig.props) i lokaciju na koju će se snimiti (mi smo odabrali vršni direktorij solutiona), te kliknite Add. Potom se vratite u panel Property Managera, kliknite na strelicu pokraj imena projekta, potom na strelicu pokraj mape Debug te dvokliknite na naziv vašeg property sheeta (u ovom primjeru to je GLConfig).



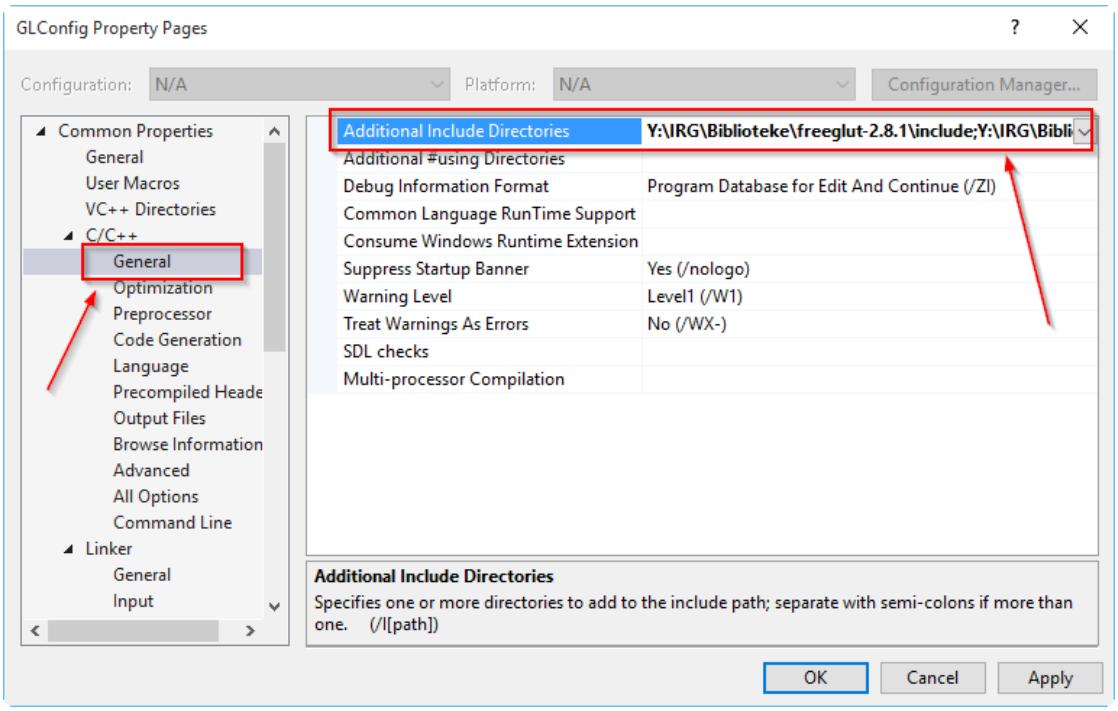
Slika A.7. Stvaranje vlastitog Property sheeta GLConfig.props

Otvorit će se prozor kao na sljedećoj slici:



Slika A.8. Podešavanje postavki

Sada ćemo podesiti sve potrebne parametre kako bi se naši primjeri mogli ispravno izgraditi i pokrenuti. Započinjemo odabirom kartice 'General' unutar grupe 'C/C++', kao na slici dolje:



Slika A.9. Podešavanje staza do biblioteka

Pod Additional Include Directories potrebno je dodati sljedeće direktorije:

Y:\IRG\Biblioteke\glm

Y:\IRG\Biblioteke\freeglut-2.8.1\include

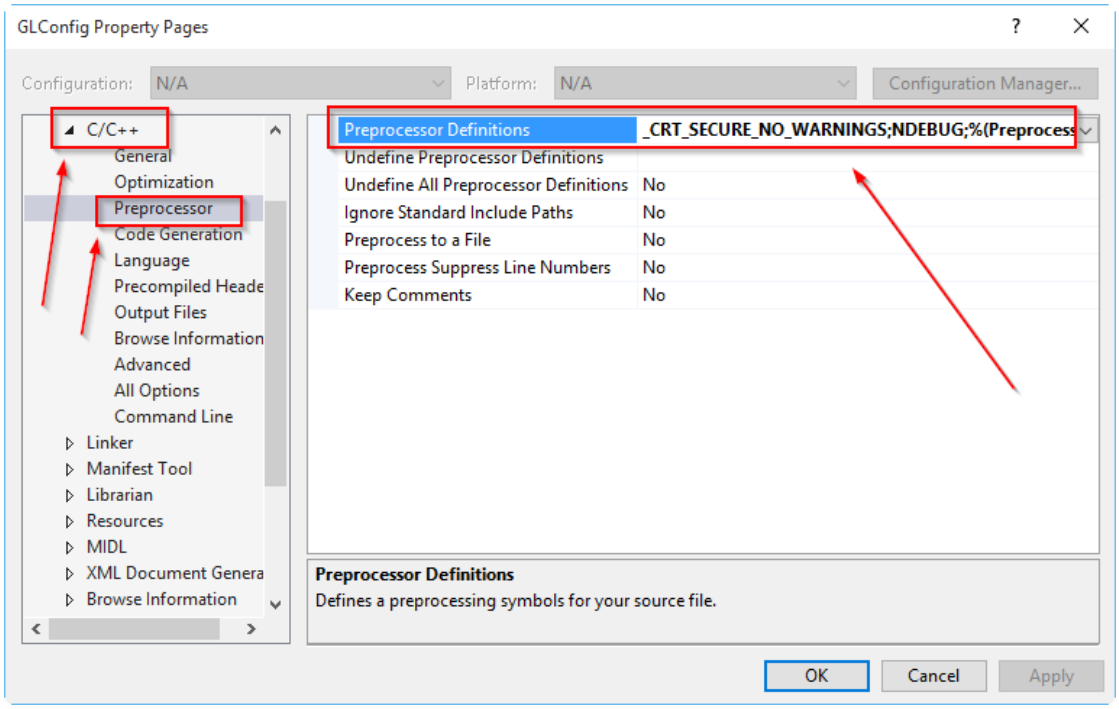
Y:\IRG\Biblioteke\glew-1.13.0\include

Ovdje Y:\ zamijenite vršnim direktorijem u koji ste raspakirali direktorij IRG. Ako ručno upisujete direktorije odvojite ih točkom zarez. Alternativno, u padajućem izborniku odaberite opciju <Edit…> pa ih dodajte preko sučelja koje će se pojaviti.

Potom odaberite karticu Preprocessor, te pod Preprocessor Definitions dodajte sljedeće:

_CRT_SECURE_NO_WARNINGS;NDEBUG

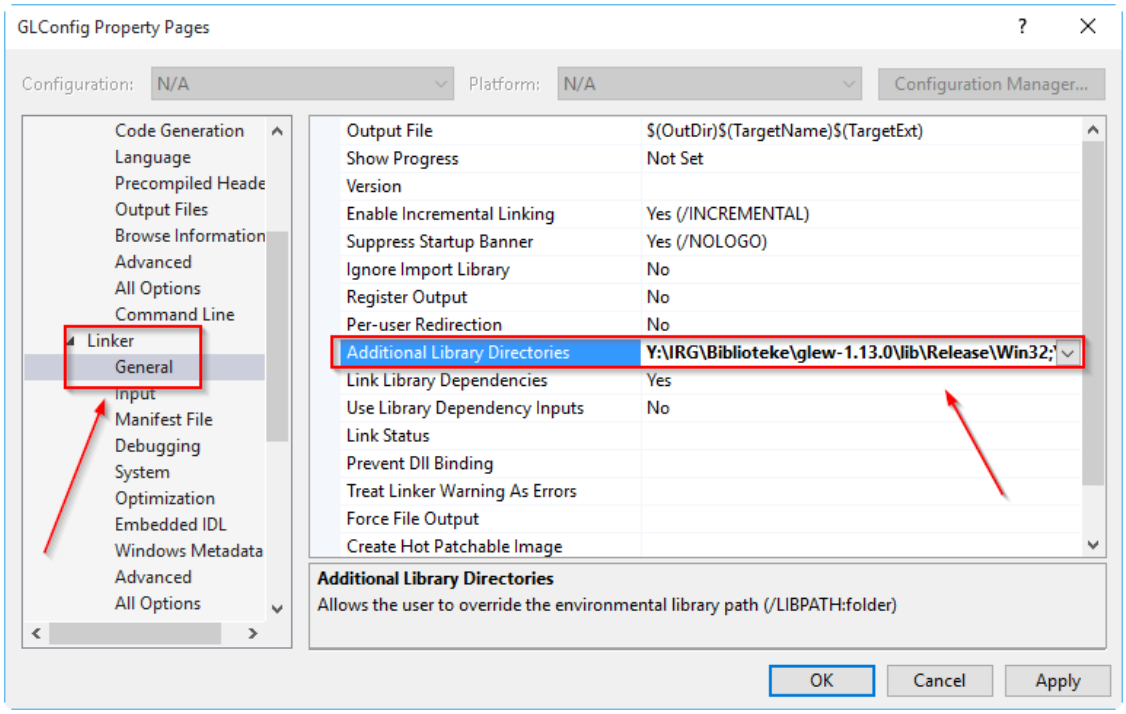
kao što je ilustrirano sljedećom slikom:



Slika A.10. Podešavanje Preprocessor Definitions

U slučaju da je u toj liniji već bilo nešto napisano (kao na slici gore) pazite da dodate točku zarez nakon NDEBUG kako bi svi elementi bili ispravno odvojeni.

Potom otvorite karticu Linker i unutar nje odaberite General.



Slika A.11. Podešavanje Additional Library Directories

Pod Additional Library Directories potrebno je unijeti sljedeće:

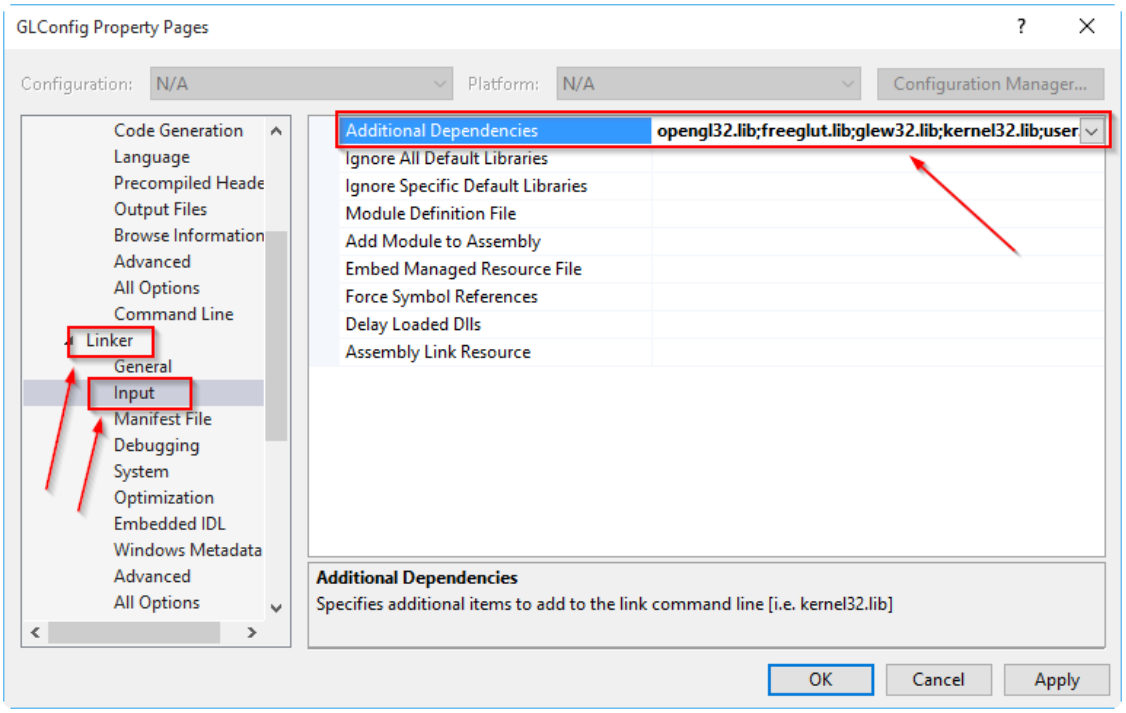
Y:\IRG\Biblioteke\glew-1.13.0\lib\Release\Win32

Y:\IRG\Biblioteke\freeglut-2.8.1\lib\x86\Debug

Ovdje Y:\ zamijenite vršnim direktorijem u koji ste raspakirali direktorij IRG. Ako ručno upisujete direktorije odvojite ih točkom zarez. Alternativno, u padajućem izborniku odaberite opciju <Edit…> pa ih dodajte preko sučelja koje će se pojaviti.

Sada pod karticom Linker odaberite Input. Ispred svih ostalih biblioteka u liniji Additional Dependencies dodajte:

opengl32.lib;freeglut.lib;glew32.lib;



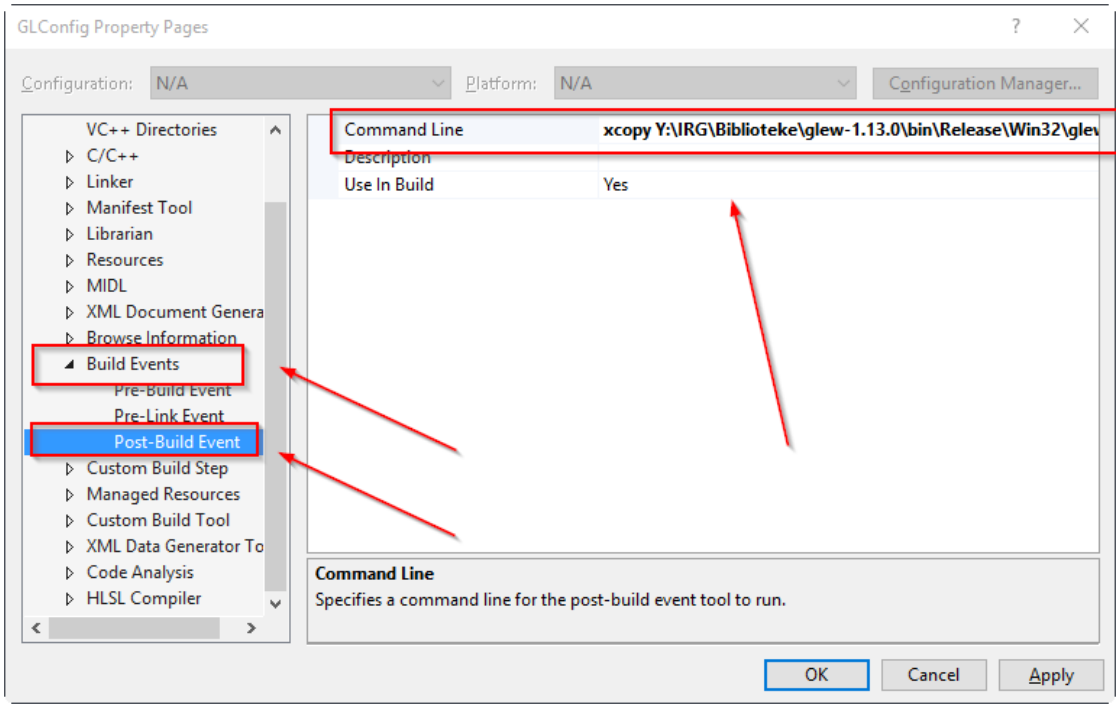
Slika A.12. Podešavanje Additional Dependencies

Jednom kad izgradimo naš program, za njegov ispravan rad bit će potrebno da se u njegovom direktoriju nalaze DLL-ovi za freeglut i glew. Kako ne bismo morali te DLL-ove svaki put ručno kopirati, definirat ćemo da se to automatski radi pri svakom buildu. Odaberite karticu Build Events, te pod Post-Build Event u liniji Command Line dodajte sljedeće (ilustrirano na slici dolje):

xcopy "Y:\IRG\Biblioteke\glew-1.13.0\bin\Release\Win32\glew32.dll" "$(OutputPath)" /Y

xcopy "Y:\IRG\Biblioteke\freeglut-2.8.1\lib\x86\Debug\freeglut.dll" "$(OutputPath)" /Y

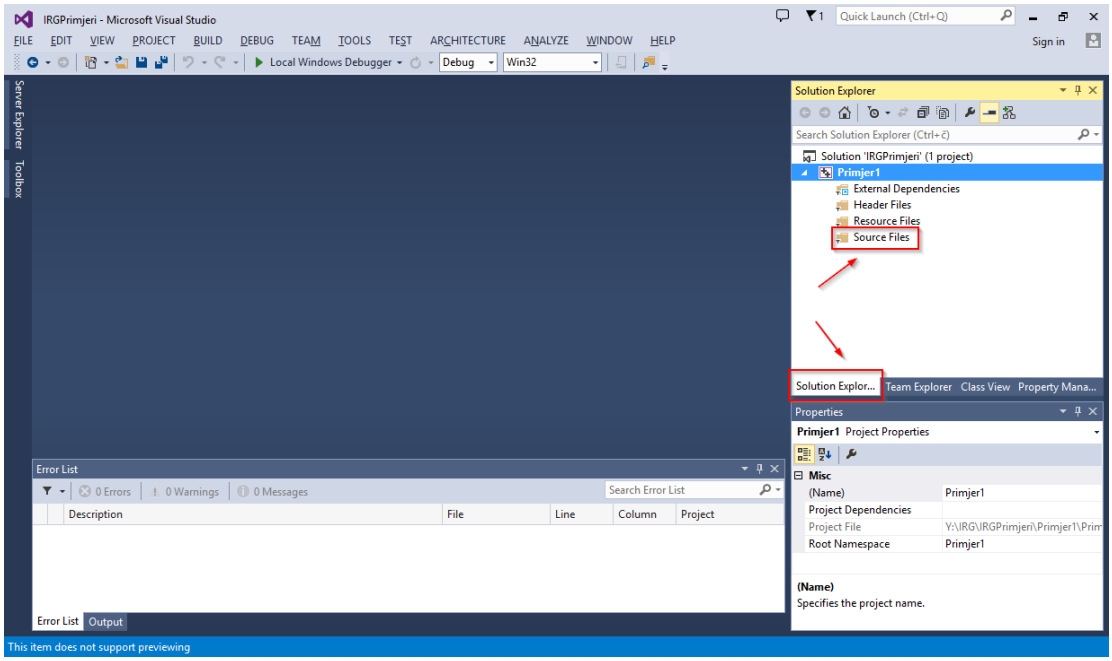
Naravno, i ovdje je potrebno Y:\ zamijeniti vršnim direktorijem u koji ste raspakirali IRG. Navodnici nisu potrebni ako vaša putanja Y:\ nema razmaka.



Slika A.13. Kopirenje DLLova za freeglut i glew

Kliknite OK, i time je postupak konfiguracije završen. Konfiguracija je sada spremljena u property sheet GLConfig.props i možete je koristiti za svaki sljedeći projekt. Ujedno, ona je automatski pridijeljena trenutno aktivnom projektu (u našem slučaju to je projekt Primjer1).

Sada možemo dodati CPP datoteke s kodom u naš projekt. Vratite se natrag na Solution Explorer tab u desnom gornjem panelu, kao što je ilustrirano na slici dolje.



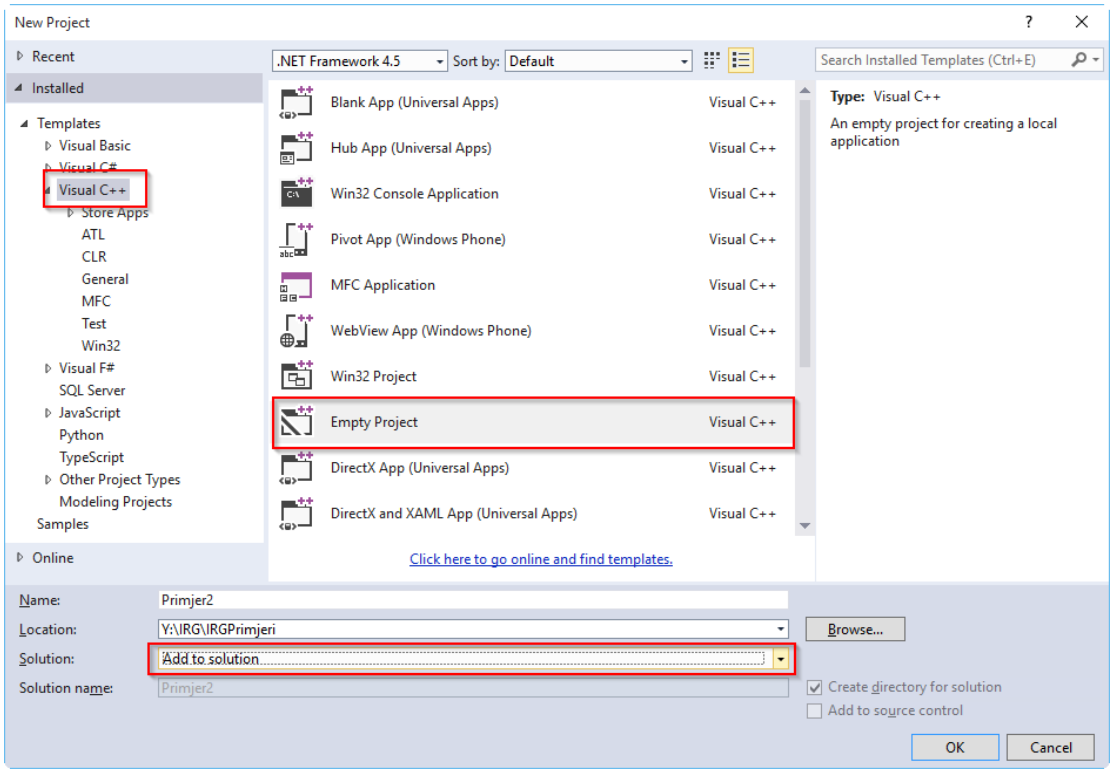
Slika A.14. Povezivanje s Primjerom 1 uz korištenje konfiguracije GLConfig.props

Desnim klikom na Source Files dobit ćete opciju Add → Existing item… Sada možete dodati datoteku Y:\IRG\Primjeri\primjer01\prozor.cpp

Pritiskom na F5 primjer bi se trebao ispravno izgraditi i moći pokrenuti.

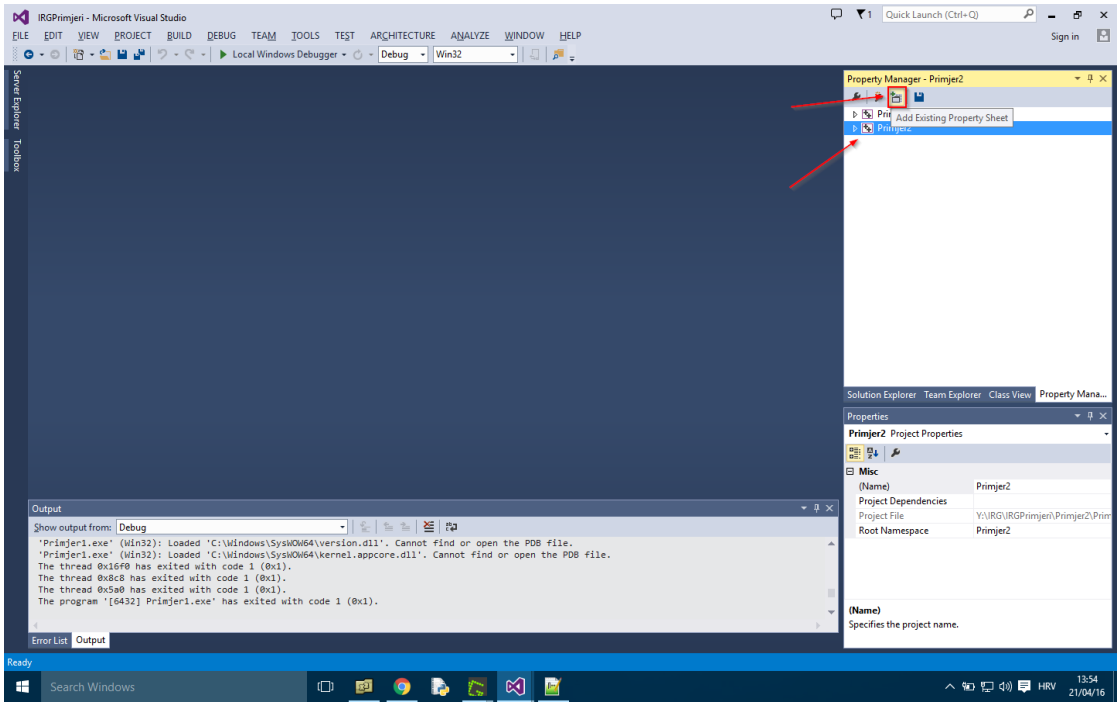
Sada možete dodati projekte i za ostale primjere koji vas zanimaju. S obzirom da svaki primjer definira metodu main, preporuča se da svaki primjer stavite u poseban projekt. Alternativa je da sve dodate u isti projekt, pa da svaki put zakomentirate sve main metode osim one koju trenutno koristite.

Dodajte novi projekt za primjer 2 tako da odaberete opcije kao na slici dolje:



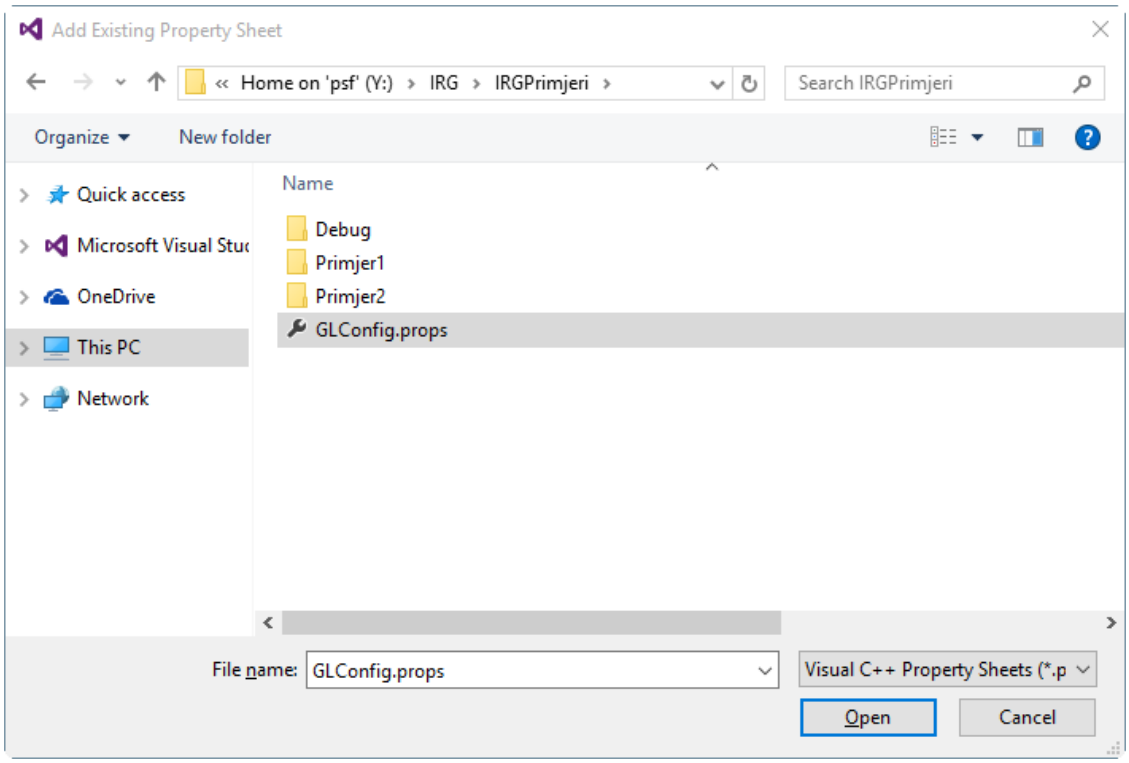
Slika A.15. Stvaranje novog projekta

Uštedjet ćemo vrijeme potrebno za konfiguriranje svih biblioteka tako da ćemo novi projekt povezati s postojećim property sheetom koji smo napravili u prethodnim koracima. Da biste to napravili, odaberite opet View → Other windows → Property Manager. U Property Manageru selektirajte svoj novi projekt (ovdje je to drugi redak u popisu, Primjer 2). Potom kliknite na gumb Add Existing Property Sheet (označen na slici).



Slika A.16. Povezivanje stvorenog projekta s Property Sheet

Otvorit će se dijalog za odabir property sheeta. Odaberite mjesto gdje ste snimili property sheet i kliknite Open.



Slika A.17. Povezivanje stvorenog projekta s prije načinjenim GLConfig.props

Sva svojstva su sada učitana iz spremljenog property sheeta, što možemo provjeriti desnim klikom na ime projekta u Solution Exploreru i odabirom Properties. Vaš novi projekt odmah je spreman za pisanje novog ili učitavanje postojećeg koda koji ovisi o bibliotekama freeglut, glew i glm. Ovaj property sheet sada možete koristiti unutar bilo kojeg drugog projekta ili solutiona.

Važna napomena vezana uz primjere: od primjera 2b nadalje primjeri učitavaju programe za sjenčare iz datoteka, i to sljedećim pozivom:

```
programID = loadShaders("SimpleVertexShader.vertexshader",
"SimpleFragmentShader.fragmentshader");
```

Uočite: ovdje nije definirana putanja do datoteka SimpleVertexShader.vertexshader i SimpleFragmentShader.fragmentshader. Da bi vaš program ispravno radio, potrebno je da ili u ovu liniju dodate punu putanju do vaših datoteka (koristite dvostruku obrnutu kosu crtu, \\) ili da datoteke iskopirate u direktorij u kojem se nalazi izvršna datoteka vašeg programa.

DODATAK B

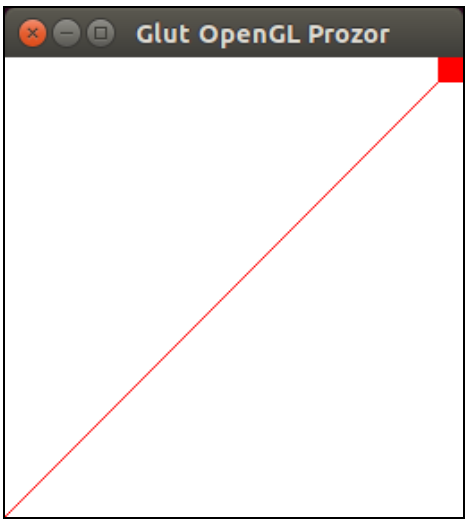
Ovaj dodatak sadrži pojašnjenja programskih primjera sa sjenčarima za OpenGL inačice 3 na više, dostupnih na stranicama kolegija. Dodatak je strukturiran tako da je za svaki primjer dan kratki opis te su uvedeni osnovni pojmovi potrebni za njegovo razumijevanje. Primjeri su zamišljeni tako da se kroz njih prolazi slijedno počevši od prvog; savjetujemo studentima da ne preskaču primjere.

Za prevođenje i pokretanje ovih primjera pod operacijskim sustavom Windows korištenjem razvojnog okruženja Microsoft Visual Studio potrebno je proučiti Dodatak A ovog dokumenta. Za prevođenje pod operacijskim sustavom Linux uz svaki primjer nalazi se i bash skripta `prozor.sh`. Primjer možete prevesti tako da u konzoli napišete `bash prozor.sh`.

Primjer 1.

Primjer 1 je osnovni OpenGL primjer s GLUT-om u kojem se otvara prozor i iscrtava jedna linija te jedan kvadratić. Za iscrtavanje koristimo klasične OpenGL pozive (OpenGL inačica 2). Primjer je napisan u skladu s osnovnim primjerom danim u knjizi (poglavlje 1) Interaktivna računalna grafika kroz primjere u OpenGL-u (http://www.zemris.fer.hr/predmeti/irg/knjiga.pdf).

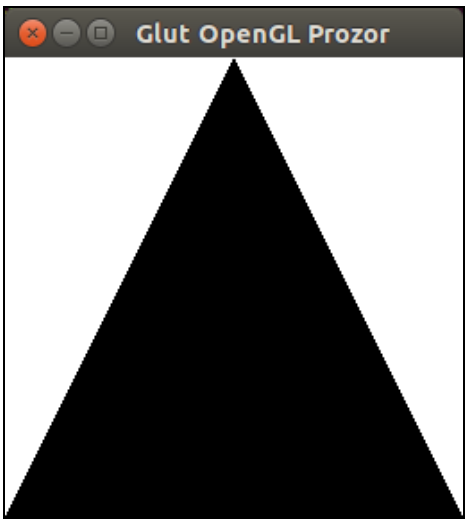
Pokretanjem primjera iscrtat će se sljedeće:



Slika B.1. Crtanje linje

Primjer 2.

Primjer 2 je osnovni primjer u kojem za iscrtavanje koristimo sjenčare, odnosno OpenGL inačice 3. Ovaj primjer na ekranu iscrtava trokut korištenjem pretpostavljene funkcionalnosti sjenčara vrhova i sjenčara fragmenata koji svaki vrh ostavljaju nepromijenjenim i iscrtavaju točke crnom bojom. Pokretanjem programa dobit će se slika prikazana u nastavku:



Slika B.2. Crtanje trokuta

Metode koje su ključne za razumijevanje ovog primjera su metoda `main`, metoda `init_data` i metoda `myDisplay`.

2.1. Metoda `main`

Metoda `main` započinje inicijalizacijom OpenGL-a:

```
glutInitContextVersion(3, 3);

glutInitContextProfile(GLUT_CORE_PROFILE);
```

Metoda `glutInitContextVersion` odabire inačicu OpenGLa koja će se koristiti u aktivnom OpenGL kontekstu. OpenGL kontekst možemo promatrati kao podatkovnu strukturu u kojoj se pamte sva stanja i podaci jednog primjerka OpenGLa. Kontekst se implicitno stvara kada stvaramo OpenGL prozor. U prikazanom primjeru metodom `glutInitContextVersion` definirano je da će se koristiti OpenGL verzije 3.3.

Metoda `glutInitContextProfile` postavlja profil koji će se koristiti u OpenGL kontekstu. Profili su podskupovi funkcionalnosti OpenGL-a korisni za

različite primjene (razvoj igara, računalno modeliranje i sl.). U ovom primjeru učitan je `GLUT_CORE_PROFILE`, čime imamo pristup podskupu osnovnih funkcionalnosti OpenGL-a.

Daljnja inicijalizacija odvija se na način uobičajen u ranijim inačicama OpenGL-a: funkcijom `glutInitDisplayMode` definira se način prikaza, potom se inicijalizira i stvara prozor te se definiraju funkcije za prikaz, promjenu veličine prozora, te obradu pritiska miša i tipkovnice. Konačno, poziva se:

```
glewExperimental = GL_TRUE;

glewInit();
```

Ovim se pozivom inicijalizira biblioteka GLEW koja nam omogućuje da na jednostavan način koristimo moderne API funkcije OpenGL-a. Dostupnost pojedinih funkcija OpenGL-a ovisi o konkretnom sklopovlju na kojem se program izvodi. Bez GLEW-a, u našem bismo programu morali ručno provjeravati koja je verzija OpenGL-a podržana trenutnim grafičkim sklopovljem, koje funkcije su podržane, te bismo potom za svaku funkciju koju želimo koristiti morali inicijalizirati pokazivač. GLEW značajno pojednostavljuje ovaj proces. Da bismo omogućili korištenje GLEW-a u našem programu sve što je potrebno je pozvati funkciju `glewInit`. Linija `glewExperimental = GL_TRUE` osigurava da GLEW ispravno radi i u slučaju da se koriste eksperimentalni ili neslužbeni upravljački programi za grafičku karticu, što može biti slučaj npr. pod operacijskim sustavom Linux.

Nakon što je završena incijalizacija konteksta i GLEW-a, metoda `main` poziva korisničku metodu `init_data` kojom se inicijaliziraju sve strukture i podaci koji će se koristiti u programu. U slučaju da su podaci uspješno inicijalizirani, pokreće se iscrtavanje pozivom metode `glutMainLoop`.

2.2. Metoda `init_data`

U metodi `init_data` inicijaliziramo potrebne podatkovne strukture kojima ćemo osigurati iscrtavanje jednog trokuta na ekran. Tri su pojma važna za razumijevanje metode: VertexArrayObject, VertexBufferObject i tok vrhova.

2.2.1 VertexArrayObject

`VertexArrayObject` (VAO) enkapsulira varijetet informacija o nizu vrhova: koordinate, normale, boje, redoslijed kojim se polje vrhova pretvara u tok vrhova... Objekti tipa `VertexArrayObject` stvaraju se naredbom `glGenVertexArrays`. Naredba u memoriji stvara traženi broj ovih objekata, za svaki objekt generira jedinstveni identifikator koji se kasnije koristi za referenciranje

pojedinog objekta, te pozivatelju u predano polje vraća popis tih identifikatora. Identifikatori su tipa `GLuint`. U najjednostavnijem slučaju predano polje može biti veličine 1 čime se umjesto adrese polja može predati adresa obične varijable tipa `GLuint`.

U ovom primjeru, na početku metode `init_data` sljedećim je pozivom zatraženo stvaranje jednog VAO-a:

```
glGenVertexArrays(1, &vertexArrayID);
```

Identifikator stvorenog VAO-a upisao se u varijablu `vertexArrayID`. Uočite da je u ovom primjeru varijabla `vertexArrayID` tretirana kao jednoelementno polje i stoga je predana njezina adresa.

Naredba kojom se VAO postavlja kao trenutni je `glBindVertexArray`. U našem primjeru, postavljamo upravo stvoreni VAO kao trenutni pozivom:

```
glBindVertexArray(vertexArrayID);
```

`VertexArrayObject` opisuje *koje sve podatke* imamo o nekom nizu vrhova, ali *ne sadržava eksplicitno te podatke*, već referencira druge objekte (tipa `VertexBufferObject`) koji ih čuvaju.

2.2.2. VertexBufferObject

`VertexBufferObject` (VBO) predstavlja jedan niz podataka u memoriji; to može biti polje koordinata, isprepleteno (engl. *interlaced*) polje koordinata i boja, polje indeksa koje određuje kojim se redoslijedom dohvaćaju vrhovi, itd.

Objekti tipa `VertexBufferObject` stvaraju se pozivom metode `glGenBuffers`:

```
glGenBuffers(kolikoSpremnikaTrebaStvoriti, &identifikatorSpremnika)
```

pri čemu se može tražiti stvaranje jednog ili više objekata, a naredba u predano polje identifikatora za svaki stvoreni objekt upisuje jedinstveni identifikator koji se kasnije koristi za referenciranje tog objekta. U ovom primjeru zatraženo je stvaranje jednog spremnika čiji se identifikator upisuje u varijablu `vertexbuffer`:

```
glGenBuffers(1, &vertexbuffer);
```

Nakon što je spremnik stvoren, potrebno je definirati koju će ulogu taj spremnik imati u programu, tj. koji tip podataka će čuvati. Ovo radimo pozivom metode `glBindBuffer` koja ima sljedeći potpis:

```
void glBindBuffer(GLenum target, GLuint buffer);
```

Prvi argument označava vrstu spremnika, a drugi argument predstavlja jedinstveni identifikator spremnika koji je vratila metoda `glGenBuffers`.

Važno: Metoda `glBindBuffer` određuje da se za zadanu vrstu spremnika u trenutnom VAO koristi spremnik čiji je identifikator drugi argument. Jedan VAO u jednom trenutku može za istu vrstu spremnika koristiti jedan podatkovni spremnik. Poseban slučaj je vrsta `GL_ARRAY_BUFFER` koja nije vezana uz trenutni VAO, već se ponaša kao "globalna" varijabla.

U ovom primjeru poziva se:

```
glBindBuffer(GL_ARRAY_BUFFER, vertexbuffer);
```

U ovom primjeru vrsta je `GL_ARRAY_BUFFER`, što označava da će naš spremnik čuvati podatke o vrhovima koji će kasnije biti obrađeni odgovarajućim primitivom za iscrtavanje. Kao što je već rečeno, `GL_ARRAY_BUFFER` se ponaša kao "globalna" varijabla na razini programa; ovim pozivom definirat ćemo da je trenutan spremnik s podacima o vrhovima sadržan u spremniku čiji je identifikator pohranjen u varijabli `vertexbuffer`. Za razliku od vrste `GL_ARRAY_BUFFER`, sve ostale vrste spremnika bit će vezane uz trenutni VAO (onaj kojeg smo zadnje postavili kao trenutni pozivom `glBindVertexArray(…)`).

Kako bismo spremnik vrhova napunili podacima, potrebno je pozvati funkciju `glBufferData`. U ovom primjeru se poziva:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(g_vertex_buffer_data),
g_vertex_buffer_data, GL_STATIC_DRAW);
```

Pri tome je `g_vertex_buffer_data` polje s podacima koje je korisnik pripremio i koje je potrebno prekopirati u aktivni spremnik vrste `GL_ARRAY_BUFFER`. Kod metode `glBufferData` prvi argument određuje vrstu spremnika u koji je potrebno pohraniti podatke (koji je to točno spremnik, prethodno je trebalo biti postavljeno pozivom `glBindBuffer`), drugi argument ukupnu veličinu podataka u oktetima koje je potrebno prekopirati u spremnik, treći argument predstavlja adresu od koje se u korisnikovoj memoriji nalaze pripremljeni podaci, a četvrti argument definira način uporabe ovih podataka (koristimo `GL_STATIC_DRAW`).

2.3. Tok vrhova

Jednom kada je spremnik vrhova definiran, kako bi se pokrenulo iscrtavanje potrebno je poslati tok (stream) vrhova sjenčaru vrhova na obradu. Program sjenčara vrhova definira niz atributa koje prima za svaki vrh. Atributi su numerirani od 0 na više. Primjerice, program sjenčara može tražiti da kao nulti atribut dobije koordinate vrha a kao prvi atribut boju vrha.

Kad se pokrene program sjenčara vrhova, za svaki vrh, od svih atributa koji su poznati za taj vrh, programu se dostavljaju samo atributi koji su omogućeni. Omogućavanje slanja atributa se radi s:

```
glEnableVertexAttribArray(indeksAtributa);
```

a onemogućavanje s:

```
glDisableVertexAttribArray(indeksAtributa);
```

U općem slučaju, spremnik s podacima može za svaki vrh sadržavati različite informacije. U najjednostavnijem slučaju, spremnik s podacima sadrži samo slijedno zapisane koordinate vrhova. U složenijem slučaju, spremnik može sadržavati isprepleteno najprije koordinate prvog vrha, pa boju prvog vrha, potom koordinate drugog vrha, pa boju drugog vrha itd. Stoga je potrebno definirati na koji će se način iz spremnika rekonstruirati određeni atributi.

Kako se to točno radi za određeni atribut, određuje se metodom `glVertexAttribPointer(indeks,…)`. Ova metoda u VAO upisuje da se atribut rednog broja `indeks` dohvaća iz spremnika koji je trenutno vezan pod `GL_ARRAY_BUFFER` na način definiran ostatkom argumenata. Kako se ovo zapisuje u VAO, korisnik kasnije pod `GL_ARRAY_BUFFER` može povezati neki drugi spremnik i to ništa ne mijenja - i dalje se za zadani atribut koristi spremnik koji je bio "trenutni" u trenutku poziva ove metode.

Nakon što korisnik stvori jedan VAO i postavi ga kao aktivnog, može stvoriti jedan ili više VBO-ova (gdje svaki čuva određenu vrstu informacija) te potom funkcijom `glVertexAttribPointer` pojasniti na koji se način za trenutni VAO podatci za atribut zadanog indeksa dohvaćaju iz trenutnog VBO-a. Ta se informacija pohranjuje u sam VAO.

Potpis metode `glVertexAttribPointer` je sljedeći:

```
void glVertexAttribPointer(
      GLuint index,
      GLint size,
      GLenum type,
      GLboolean normalized,
      GLsizei stride,
      const GLvoid * pointer);
```

U ovom primjeru, niz `g_vertex_buffer_data` sadrži 9 elemenata i predstavlja tri vrha s po tri koordinate. Kako bi se sjenčaru slale tri po tri koordinate, definiramo:

```
glVertexAttribPointer(
```

```
          0,  // indeks atributa, koji mora odgovarati indeksu u
sjenčaru

          3,  // broj koordinata u vrhu - mora biti 1, 2, 3 ili 4

          GL_FLOAT, // tip koordinate

          GL_FALSE, // automatska normalizacija na raspon [-1, 1]

          0,        // razmak u oktetima između vrhova; 0: slijedni
vrhovi

          (void*)0  // indeks okteta u nizu na kojem počinju vrhovi

      );
```

Predzadnji parametar (`stride`) predstavlja ukupan broj okteta između početaka uzastopnih zapisa atributa. U ovom primjeru spremnik sadrži koordinate vrhova koje su tipa GL_FLOAT (4 okteta); kako svaki vrh ima tri koordinate stride bi bio 4*3=12 jer podaci o sljedećem vrhu u spremniku kreću neposredno nakon. Kada je to slučaj, stride je dozvoljeno postaviti na vrijednost 0, čime će OpenGL sam izračunati ovu vrijednost.

Pretvorba u `void*` posljednjeg parametra potrebna je zbog kompatibilnosti sa starijim verzijama OpenGL-a, no posljednji parametar predstavlja običan redni broj okteta u nizu na kojem počinju podaci o vrhovima. Kako u prikazanom slučaju podaci o vrhovima počinju od početka polja, ovaj parametar postavljen je na 0. Više informacija o ovoj metodi moguće je pronaći u dokumentaciji metode.

Po završetku metode `init_data` sve podatkovne strukture potrebne za iscrtavanje trokuta na ekran su inicijalizirane. Definiran je jedan VAO i postavljen je kao trenutni. Potom je stvoren jedan VBO i napunjen je podacima o 3 vrha. Konačno, definirano je na koji način se podaci trebaju čitati iz VBO-a (počevši od početka, čitaju se po 3 koordinate koje su decimalni brojevi, te nije potrebno raditi normalizaciju). Jednom kad su inicijalizirane podatkovne strukture, moguće je iscrtati trokut na ekran. Iscrtavanje se odvija u metodi `myDisplay`.

2.4. Metoda `myDisplay`

Metoda `myDisplay` započinje brisanjem ekrana. Nakon toga kao trenutni VAO postavlja se VAO kojeg se inicijaliziralo u metodi `init_data`. Kako bi se to postiglo, koristi se njegov identifikator pohranjen u varijabli `vertexArrayID` koji je popunjen u metodi `init_data`:

```
glBindVertexArray(vertexArrayID);
```

U metodi `init_data` metodom `glVertexAttribPointer` definirali smo vrhove kao atribut s indeksom 0 koji se sastoji od 3 koordinate. Prije no što se zatraži crtanje, potrebno je definirati koji se sve atributi šalju sjenčaru. Omogućavanje slanja atributa obavlja se naredbom `glEnableVertexAttribArray` koja kao argument dobiva redni broj atributa. U slučaju da je potrebno za svaki vrh slati više od jednog atributa, svaki od atributa omogućava se zasebno. Kako se u slučaju iz primjera koristi samo atribut s indeksom 0, odgovarajući poziv je:

```
glEnableVertexAttribArray(0);
```

gdje je kao indeks atributa predano 0, što je u skladu s ranije definiranim indeksom u funkciji `glVertexAttribPointer`.

Nakon ovoga pristupa se iscrtavanju jednostavnim pozivom:

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

čime je zatraženo iscrtavanje **toka vrhova** pri čemu se taj tok tretira kao trokute (definirano konstantom `GL_TRIANGLES`). Potpis metode `glDrawArrays` je sljedeći:

```
void glDrawArrays(GLenum mode, GLint first, GLsizei count);
```

Parametar `mode` određuje koji primitiv želimo crtati korištenjem vrhova. Primjeri podržanih primitiva opisani su u knjizi i obrađeni na predavanjima; neki od njih su `GL_TRIANGLES`, `GL_POINTS`, `GL_LINE_STRIP`, `GL_LINE_LOOP`, `GL_LINES`, `GL_TRIANGLE_STRIP`, itd. Parametar `first` određuje indeks prvog elementa u svakom omogućenom toku koji će se obrađivati, dok parametar `count` određuje koliko će se elemenata obraditi. U ovom primjeru, omogućeno je slanje atributa 0 (trojki koordinata vrhova), te je pozivom metode `glDrawArrays(…)` zatraženo da se za dani tok počevši od indeksa 0 uzme 3 vrha (točno toliko ih i imamo u toku) te da se korištenjem ta 3 vrha iscrta trokut.

Ako se crtanje radi metodom `glDrawArrays(…)`, tok vrhova generira se izravno prema redoslijedu kojim su vrhovi spremljeni u spremnik vrhova. Ako je u VAO povezan spremnik indeksa kao GL_ELEMENT_ARRAY_BUFFER, metodom `glDrawElements(…)` može se tražiti generiranje i obrada toka vrhova koji nastaje prema zadanim indeksima (ovo je ilustrirano u primjeru 2c).

U ovom primjeru nije posebno podešavan pogled, pa je važno znati sljedeće: bez ikakvih podešavanja, kamera je smještena u ishodište te je usmjerena prema negativnoj *z*-osi, view-up vektor je pozitivna *y*-os, volumen pogleda određen je po *x*, *y* i *z*-osima od -1 do +1; *viewport* je čitav prozor. Ovime će ishodište koordinatnog sustava (pozicija kamere) biti u centru prozora; +*x* os gleda u desno, +*y* os prema gore a +*z* os po pravilu desne ruka gleda iz ekrana prema korisniku.

Više detalja o ovoj temi može se pročitati na adresi:

https://www.opengl.org/wiki/Vertex_Specification

Primjer 2b.

Primjer 2b je proširenje primjera 2 u kojem po prvi puta radimo s vlastitim programima za sjenčare vrhova i fragmenata. Ovi se programi pišu u posebnom programskom jeziku GLSL (engl. *GL Shading Language*) koji je sintaksom sličan programskom jeziku C. Ono što je pomalo neuobičajeno je da je programe sjenčara potrebno **programski učitavati, prevoditi i povezivati** izravno kroz programski kod. Ovo je važna distinkcija u odnosu na klasične programe koji se izvode na operacijskom sustavu i koji se iz izvornog koda prevode u strojni kod kroz razvojnu okolinu ili izravno uporabom naredbenog retka. U ovom primjeru, učitavanje, prevođenje i povezivanje programa za sjenčare se radi na kraju funkcije `init_data`, pozivom:

```
programID = loadShaders("SimpleVertexShader.vertexshader",
"SimpleFragmentShader.fragmentshader");
```

Metoda `loadShaders` je metoda koju smo sami napisali, smještena je u pomoćnoj biblioteci `util.cpp` i slobodno je možete koristiti u daljnjem radu pri izradi laboratorijskih vježbi. Metodom `loadShaders` učitat će se sjenčar vrhova iz datoteke `SimpleVertexShader.vertexshader` i sjenčar fragmenata iz datoteke `SimpleFragmentShader.fragmentshader`. Ukoliko sve bude u redu, programski kod oba sjenčara će biti preveden i povezan u jedinstven program kojemu će biti dodijeljen jedinstveni identifikator koji pohranjujemo u varijablu `programID`. Naknadno referenciranje na taj program obavlja se uporabom ovog identifikatora.

Važna napomena: uz ovakav poziv metode `loadShaders` program će očekivati da su datoteke sjenčara vrhova `SimpleVertexShader.vertexshader` i sjenčara fragmenata `SimpleFragmentShader.fragmentshader` smještene u istoj mapi u kojoj je i izvršna datoteka programa. Ukoliko prevodite program korištenjem razvojne okoline Microsoft Visual Studio to neće biti slučaj, pa je potrebno promijeniti putanje do datoteka tako da su apsolutne (u putanjama koristite dvostruku obrnutu kosu crtu). Primjer:

```
programID = loadShaders("C:\\IRG\\SimpleVertexShader.vertexshader",
"C:\\IRG\\SimpleFragmentShader.fragmentshader");
```

Kako bi se omogućilo da se za iscrtavanje koriste učitani sjenčari, unutar metode `myDisplay` dodana je naredba:

```
glUseProgram(programID);
```

Datoteke sjenčara vrhova `SimpleVertexShader.vertexshader` i sjenčara fragmenata `SimpleFragmentShader.fragmentshader` su obične tekstovne datoteke koje možete otvoriti unutar razvojne okoline ili u uređivaču teksta po želji. U njima je smješten GLSL programski kod sjenčara.

2.b.1. Datoteka SimpleVertexShader.vertexshader

```
#version 330 core


// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;


void main(){
    gl_Position.xyz = vertexPosition_modelspace;
    gl_Position.w = 1.0;
}
```

Program za sjenčar vrhova je vrlo jednostavan. Započinje deklaracijom verzije OpenGL-a za koju je sjenčar pisan. Potom se deklariraju ulazni podatci. Poziv `layout(location=0)` znači da želimo nulti atribut iz ulaznog spremnika (podsjetimo se, ranije je funkcijom `glVertexAttribPointer` definirano da su vrhovi koje šaljemo sjenčaru atribut indeksa 0); `in` označava da je podatak ulazni, `vec3` označava da je podatak vektor od 3 elementa, a `vertexPosition_modelspace` je varijabla u koju će podatak biti pohranjen.

U programu za sjenčar vrhova postoji jedna implicitno deklarirana varijabla `gl_Position`, i u nju program treba zapisati konačnu poziciju vrha. Naš jednostavni sjenčar samo prosljeđuje ulaz na izlaz: ulazni vrh nalazi mu se u varijabli `vertexPosition_modelspace`, i ta se vrijednost pridjeljuje varijabli `gl_Position`. Vrijednost homogene koordinate postavlja se na 1.0.

Program može deklarirati i dodatne varijable ako želi, i to kao `uniform` varijable ili pak kao `out` varijable. Varijable tipa `uniform` su globalne GLSL varijable kojima se prenose parametri iz glavnog programa u sjenčar (primjerice, programu koji će obavljati transformaciju pogleda na ovaj ćemo način predati transformacijsku matricu). Varijable tipa `out` su izlazne varijable. Vrijednosti koje se zapišu u `out` varijable interpoliraju se između vrhova i šalju u sjenčar fragmenata gdje ih se može dohvatiti tako da se deklarira varijabla tipa `in` istog imena. Ovo ćemo vidjeti kasnije u primjeru 4.

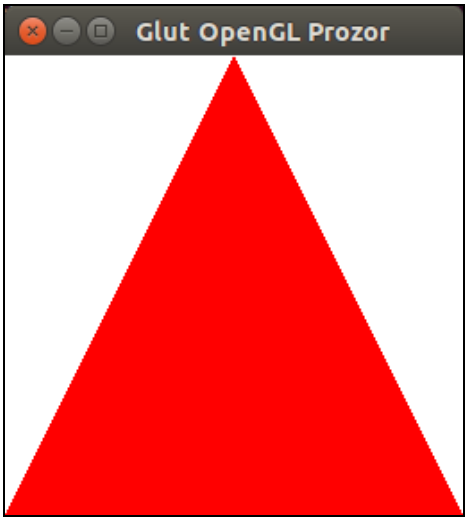
2.b.2. Datoteka SimpleFragmentShader.fragmentshader

```
#version 330 core


// Ouput data
out vec3 color;
```

```
void main()
{
	// Output color = red
	color = vec3(1,0,0);
}
```

Program za sjenčar fragmenata deklarira jednu izlaznu vrijednost tipa `vec3` te postavlja tu vrijednost na (1,0,0), što odgovara crvenoj boji. Sadržaj te varijable interpretirat će se kao boja kojom je potrebno prikazati taj fragment.

Pokretanjem ovog primjera dobit će se sljedeće:



Slika B.3. Crtanje trokuta uz korištenje sjenčara

Primjer 2c.

Primjer 2c je nadogradnja primjera 2b u kojem smo promijenili način na koji radimo s tokom vrhova. Umjesto da se tok vrhova izravno generira kao slijed vrhova iz predanog spremnika, u VAO dodajemo i spremnik indeksa kojim se vrhovi šalju kartici. Ovime iste vrhove možemo slati više puta, čime možemo ostvariti značajnu uštedu memorije i dobiti na brzini.

Prva promjena nalazi se u metodi `init_data`. Inicijalizacija VAO-a i VBO-a koji sadrži vrhove ostala je ista kao u prethodnim primjerima, ali nakon inicijalizacije VBO-a s koordinatama vrhova dodano je i stvaranje i inicijalizacija VBO-a s indeksima vrhova:

```
// Stvaramo polje za 6 indeksa

static const ushort vindices[6] = { 0, 1, 2, 0, 1, 3};



// Identifikator za indeksni spremnik

GLuint indexbuffer;

// Stvori 1 spremnik i pohrani identifikator u indexbuffer

glGenBuffers(1, &indexbuffer);

// postavi naš spremnik kao aktivni GL_ELEMENT_ARRAY_BUFFER

glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexbuffer);

// popuni naš spremnik elementima iz polja indeksa

glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(ushort)*6, vindices,
GL_STATIC_DRAW);
```

Druga promjena nalazi se u metodi `myDisplay`, gdje je metoda `drawArrays` za iscrtavanje trokuta zamijenjena metodom `drawElements`:

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, (GLvoid*)0);
```

Ovaj poziv znači da se kao tok vrhova šalju oni vrhovi čiji su indeksi specificirani spremnikom `GL_ELEMENT_ARRAY_BUFFER`, da se koristi šest indeksa počevši od nultog, te da su indeksi tipa `GL_UNSIGNED_SHORT`.

Da pojasnimo, ako se ne koristi spremnik indeksa, ako smo u spremnik vrhova povezali polje:

```
{V0,V1,V2}
```

gdje je svaki V_i primjerice tri floata, upravo se tim redoslijedom vrhovi šalju kartici, pa je tok vrhova koji se obrađuje:

```
V0 -> V1 -> V2
```

Ako s druge strane u spremnik vrhova stavimo polje:

```
{V0,V1,V2,V3}
```

tj. na poziciji 0 je V_0, na poziciji 1 je V_1, itd., te u spremnik indeksa stavimo npr:

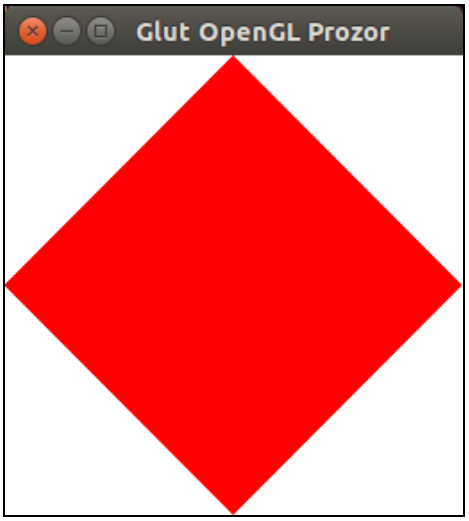
```
{0,1,2,0,1,3}
```

tok koji se šalje čine vrhovi na pozicijama određenim ovim indeksima. Stoga će kartici biti poslan tok:

```
V0 -> V1 -> V2 -> V0 -> V1 -> V3
```

U našem primjeru je primitiv za crtanje `GL_TRIANGLES`, a on grupira po tri vrha u jedan trokut, pa će se nacrtati dva trokuta: T_1 određen vrhovima V_0, V_1, V_2 te T_2 određen vrhovima V_0, V_1, V_3. Na ovaj način u polju vrhova ne moramo ponavljati vrhove. Ovu ideju koristi i OBJ datotečni format kada definira poligone.

Konačni rezultat će biti da će se iscrtati dva trokuta koja dijele bazu i slika koju ćemo dobiti je zarotirani kvadrat, kao što je prikazano na slici u nastavku.



Slika B.4. Crtanje dva trokuta uz korištenje sjenčara

Primjer 3.

U ovom primjeru počinju se koristiti matrice: radi se transformacija pogleda i perspektivna projekcija.

U program sjenčara vrhova u popis varijabli dodana je jedna matrica (varijabla je nazvana MVP) kao `uniform` varijabla (podatak koji se ne mijenja od vrha do vrha). Kako bi odredio konačni položaj, program koji je u okviru ovog primjera napisan za sjenčar vrhova će dobivenu koordinatu proširiti homogenom i pomnožiti s transformacijskom matricom MVP.

3.1. Datoteka SimpleVertexShader.vertexshader

```
#version 330 core


layout(location = 0) in vec3 vertexPosition_modelspace;

uniform mat4 MVP;


void main(){

    gl_Position = MVP * vec4(vertexPosition_modelspace, 1);

}
```

Vrijednost matrice MVP postavlja se iz **glavnog programa** (jednom, prije nego što započne obrada vrhova, i matrica ostaje nepromijenjena za sve vrhove u toj obradi). U tu svrhu, u metodi `init_data` nakon prevođenja programa sjenčara koji deklarira tu varijablu potrebno je zatražiti njezin identifikator (engl. *handle*). To se radi metodom `glGetUniformLocation` koja kao argument dobije identifikator programa te ime varijable u programu sjenčara, a vraća nam traženi identifikator koji će se kasnije koristiti kada će se htjeti u tu varijablu upisati podatak. Ovo je ilustrirano u nastavku.

```
MVPMatrixID = glGetUniformLocation(programID, "MVP");
```

Matricu koja obavlja zadanu transformaciju možemo računati ili prilikom crtanja, ili već u inicijalizaciji ako se položaj kamere ne mijenja. U ovom primjeru, izračunavanje matrice obavlja se u metodi `myDisplay`. Koriste se pomoćne metode iz biblioteke glm.

Prilikom crtanja, prije no što se pokrene samo crtanje, podatke iz izračunate matrice pozivom metode `glUniformMatrix4fv` prebacujemo u lokalnu varijablu programa sjenčara (dajemo identifikator varijable koji smo prethodno pripremili te adresu podataka koje tamo prebacujemo):

```
glUniformMatrix4fv(MVPMatrixID, 1, GL_FALSE, &mvp[0][0]);
```

Prvi parametar označava identifikator matrice, drugi koliko matrica želimo modificirati (moguće je raditi i s nizovima matrica), treći definira treba li transponirati matricu, a četvrti je adresa podataka.

Nakon ovakvog postavljanja matrice na uobičajeni način pokreće se crtanje.

Kako bi se moglo koristiti metode iz biblioteke glm, potrebno je u program uključiti zaglavne datoteke `glm/glm.hpp` i `glm/gtc/matrix_transform.hpp`. Na operacijskom sustavu Linux ove su datoteke dio biblioteke `libglm-devel` koju po potrebi treba instalirati (`sudo apt-get install ime_biblioteke`).

Pokretanjem ovog primjera iscrtat će se sljedeće:

Slika B.5. Crtanje trokuta uz korištenje sjenčara i transformacijskih matrica

Primjer 4.

U ovom primjeru za svaki od vrhova definira se boju, a OpenGL radi interpolaciju boja za svaki fragment. Proširuje se primjer 3 na način da se u metodi `init_data` uvodi novi spremnik koji će čuvati informacije o bojama (nazvan je `colorbuffer`). Taj se spremnik popunjava podacima o boji na isti način na koji se i spremnik vrhova popunjava podacima o vrhovima.

Nakon pojašnjavanja kako se iz jednog spremnika dohvaćaju pozicije (za atribut 0), pojašnjava se i kako se iz drugog spremnika dohvaćaju boje (tom atributu pridijeljen je indeks 1):

```
glBindBuffer(GL_ARRAY_BUFFER, colorbuffer);

glVertexAttribPointer(

          1,                  // attribute 1.

          3,                  // size

          GL_FLOAT,           // type

          GL_FALSE,           // normalized?

          0,                  // stride

          (void*)0            // array buffer offset

);
```

Obavezno je prije poziva metode `glVertexAttribPointer` postaviti `colorbuffer` kao trenutni `GL_ARRAY_BUFFER` pozivom metode `glBindBuffer`.

Prije iscrtavanja u metodi `myDisplay` obavezno je uključiti i slanje atributa 1 (tj. boje iz spremnika):

```
glEnableVertexAttribArray(1);
```

Sjenčar vrhova izgleda ovako:

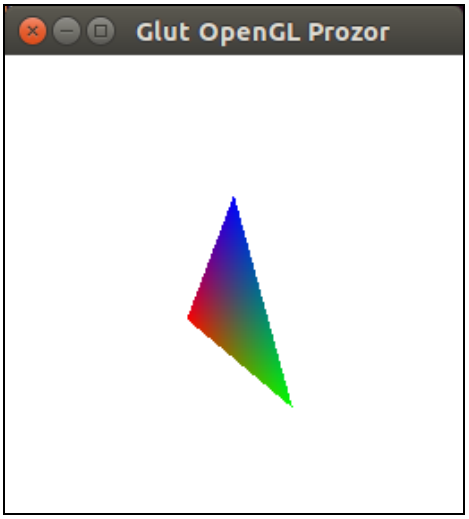
4.1. Datoteka SimpleVertexShader.vertexshader

```
#version 330 core

layout(location = 0) in vec3 vertexPosition_modelspace;

layout(location = 1) in vec3 vertexColor;
```

```
out vec3 fragmentColor;

uniform mat4 MVP;


void main(){

    gl_Position = MVP * vec4(vertexPosition_modelspace, 1);

    fragmentColor = vertexColor;

}
```

U sjenčaru vrhova dodana je lokalna varijabla tipa `in` za drugi atribut te jedna varijabla tipa `out` u koju se zapisuje primljena boja. OpenGL će za svaki od vrhova podatak koji se zapiše u tu izlaznu varijablu interpolirati po slikovnim elementima između vrhova (kada radi popunjavanje). Tu interpoliranu vrijednost možemo primiti u sjenčaru fragmenata na način da tamo deklariramo istoimenu varijablu tipa `in`. **VAŽNO: tamo stiže interpolirana vrijednost!** U ovom primjeru napisan je najjednostavniji sjenčar fragmenata koji sadržaj te varijable samo proslijedi dalje kao boju fragmenta.

Pokretanjem primjera iscrtat će se sljedeće:



Slika B.6 Interpolacija boje za svaki fragment

Primjer 5.

U ovom primjeru scena je proširena na dva trokuta koji se probadaju. Kako bi se dobio korektan prikaz uzimajući u obzir činjenicu da se trokuti probadaju, uključen je Z-spremnik.

U metodu `main` dodane su sljedeće naredbe:

```
// Omogući uporabu Z-spremnika

glEnable(GL_DEPTH_TEST);

// Prihvaćaj one fragmente koji su bliže kameri u smjeru gledanja

glDepthFunc(GL_LESS);
```

U metodi `myDisplay` obavezno je pri brisanju brisati i Z-spremnik, pa se stoga u `glClear` dodaje i konstanta `GL_DEPTH_BUFFER_BIT`:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Ostatak programa isti je kao u prethodnom primjeru. Pokretanjem programa iscrtat će se sljedeće:

Slika B.7 Upotreba Z-spremnika

Literatura

[1] Više informacija o sjenčaru vrhova može se pronaći na adresi:
OpenGL and its API: Vertex Specification
https://www.opengl.org/wiki/Vertex_Specification

[2] Prikazani primjeri dijelom su zasnovani na primjerima s adrese:
OpenGl Tutorial: Basic OpenGL
http://www.opengl-tutorial.org/beginners-tutorials/