Programowanie 3W grafiki w OpenGL. Shadery Geometrii. Krzywe Béziera

Aleksander Denisiuk
Polsko-Japońska Akademia Technik Komputerowych
Wydział Informatyki w Gdańsku
ul. Brzegi 55
80-045 Gdańsk

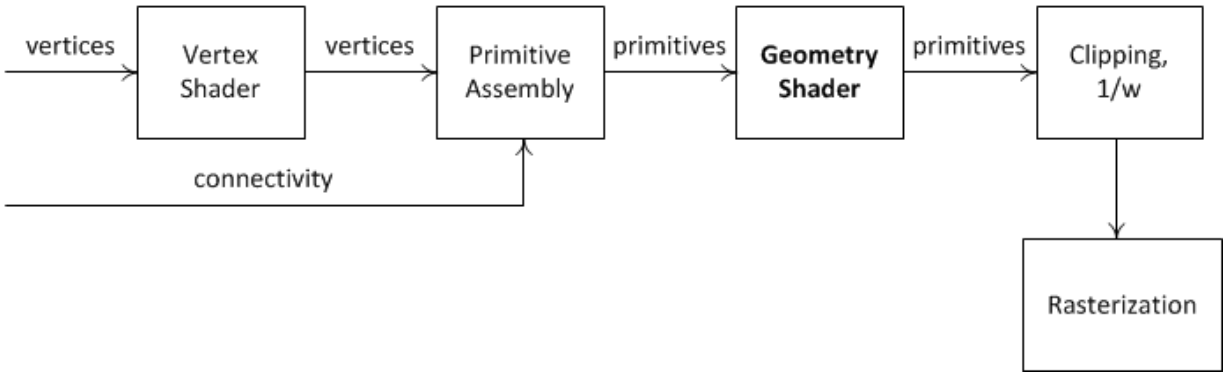
denisjuk@pja.edu.pl

Najnowsza wersja tego dokumentu dostępna jest pod adresem
http://users.pja.edu.pl/~denisjuk/

Shadery Geometrii

- Między shaderem wierzchołków a shaderem fragmentów
- Każdy prymityw graficzny
- Na wejściu prymityw graficzny (`points`, `lines`, etc)
- Na wyjściu nowy prymityw graficzny

Potok renderingu



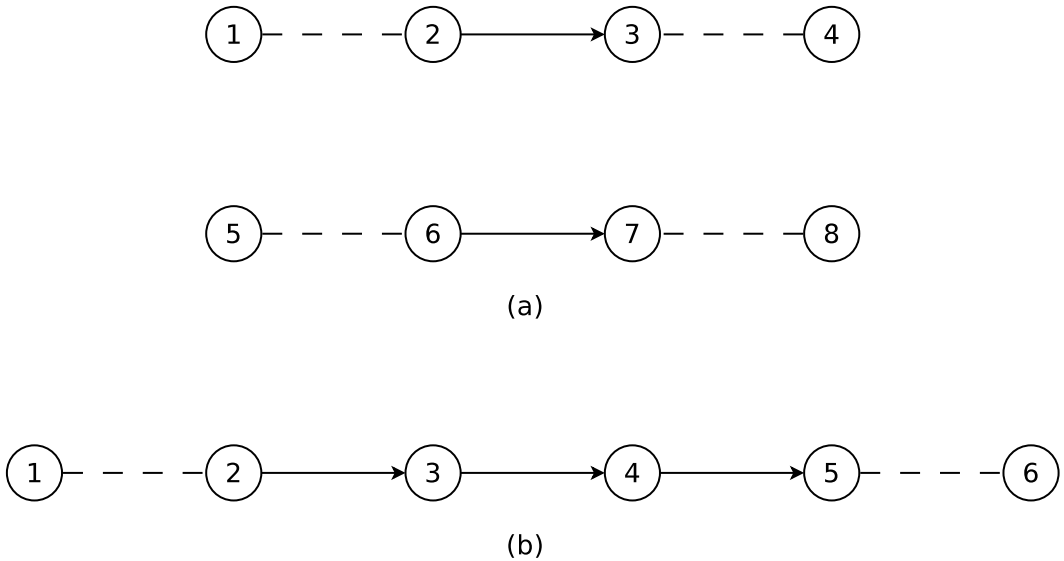


Figure 2.4. Lines with adjacency (a) and line strips with adjacency (b). The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

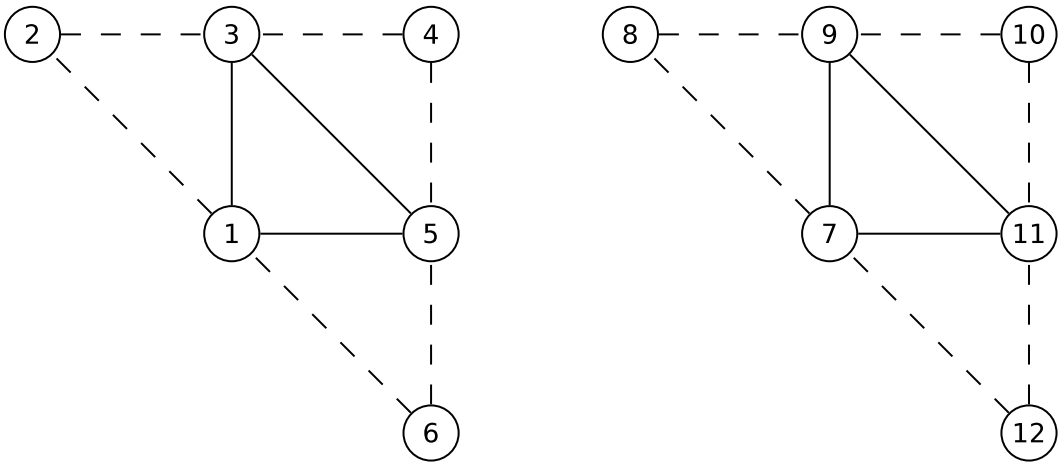


Figure 2.5. Triangles with adjacency. The vertices connected with solid lines belong to the main primitive; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

Triangle strip with adjacency

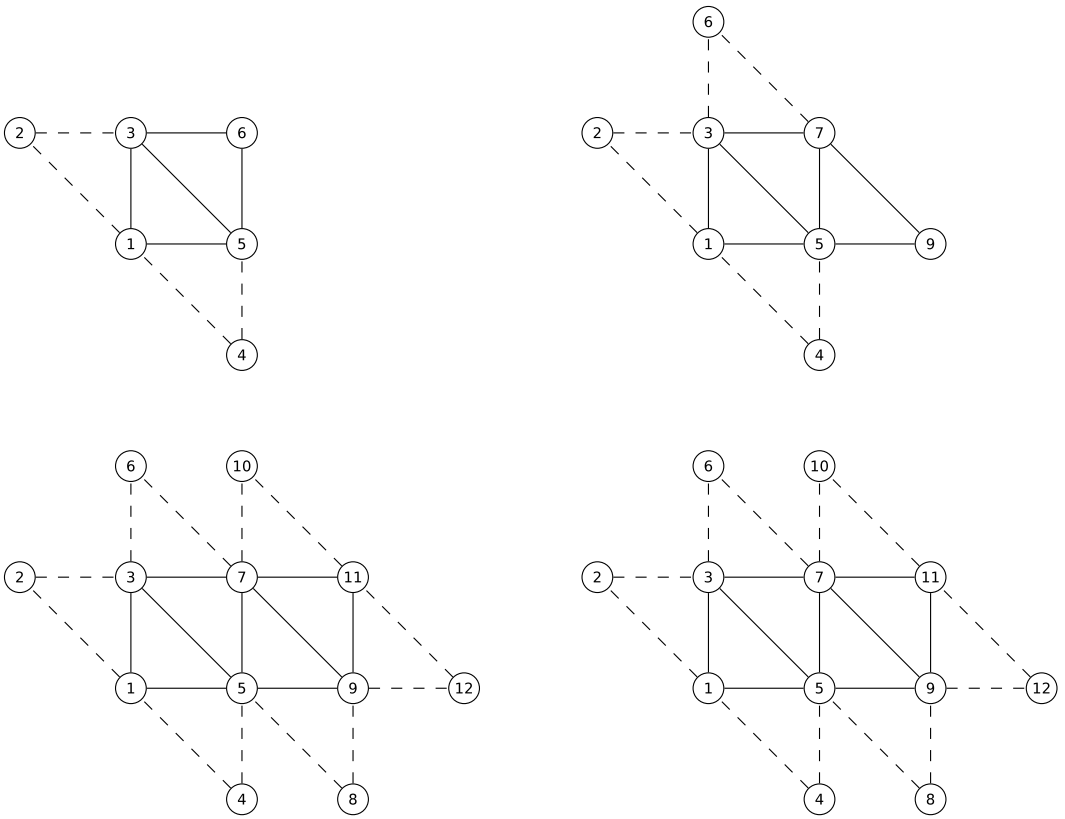


Figure 2.6. Triangle strips with adjacency. The vertices connected with solid lines belong to the main primitives; the vertices connected by dashed lines are the adjacent vertices that may be used in a geometry shader.

- points ⇐ GL_POINTS
- lines ⇐ GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP
- triangles ⇐ GL_TRIANGLES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN
- lines_adjacency ⇐ GL_LINES_ADJACENCY, GL_LINE_STRIP_ADJACENCY
- triangles_adjacency ⇐ GL_TRIANGLES_ADJACENCY, GL_TRIANGLE_STRIP_ADJACENCY
- przykład:

```
layout(triangles) in;
```

- `points`
- `line_strip`
- `triangle_strip`
- przykład:

```
layout(triangle_strip) out;
```

Maksymalna ilość wyemitowanych wierzchołków


```
layout(max_vertices=4) out;
layout(triangle_strip, max_vertices=4) out;
```

- ograniczenie (nie mniej niż 256), przykładowo 1024

```
glGetIntegerv(GL_MAX_GEOMETRY_OUTPUT_VERTICES)
```

Ilość wywołań


```
layout(invocations = 5) in;
```

- od OpenGL 4.0
- `gl_InvocationID`

Zmienne wejściowe


```
gl_in[]
gl_PrimitiveIDIn
gl_InvocationID
```

Tablica gl_in[]


```
gl_Position
gl_PointSize
gl_ClipDistance []
```

Przekazywanie informacji z shadera wierzchołków

- w shaderze wierzchołków

```
out vec3 normal;
```

- w shaderze geometrii

```
in vec3 normal[];
```

Zmienne uniform

- shader geometrii ma dostęp do zmiennych uniform

Wynik

- `gl_Position`
- `gl_PointSize`
- `gl_ClipDistance[]`
- `gl_PrimitiveID`
- `gl_Layer`
- `gl_ViewportIndex`

Wyjściowe zmienne


```
out      vec3 vertexNormal;
flat out vec3 faceNormal;
```

- `flat` nie interpoluje się

  ```
  glProvokingVertex(GL_FIRST_VERTEX_CONVENTION)
  glProvokingVertex(GL_LAST_VERTEX_CONVENTION)
  ```

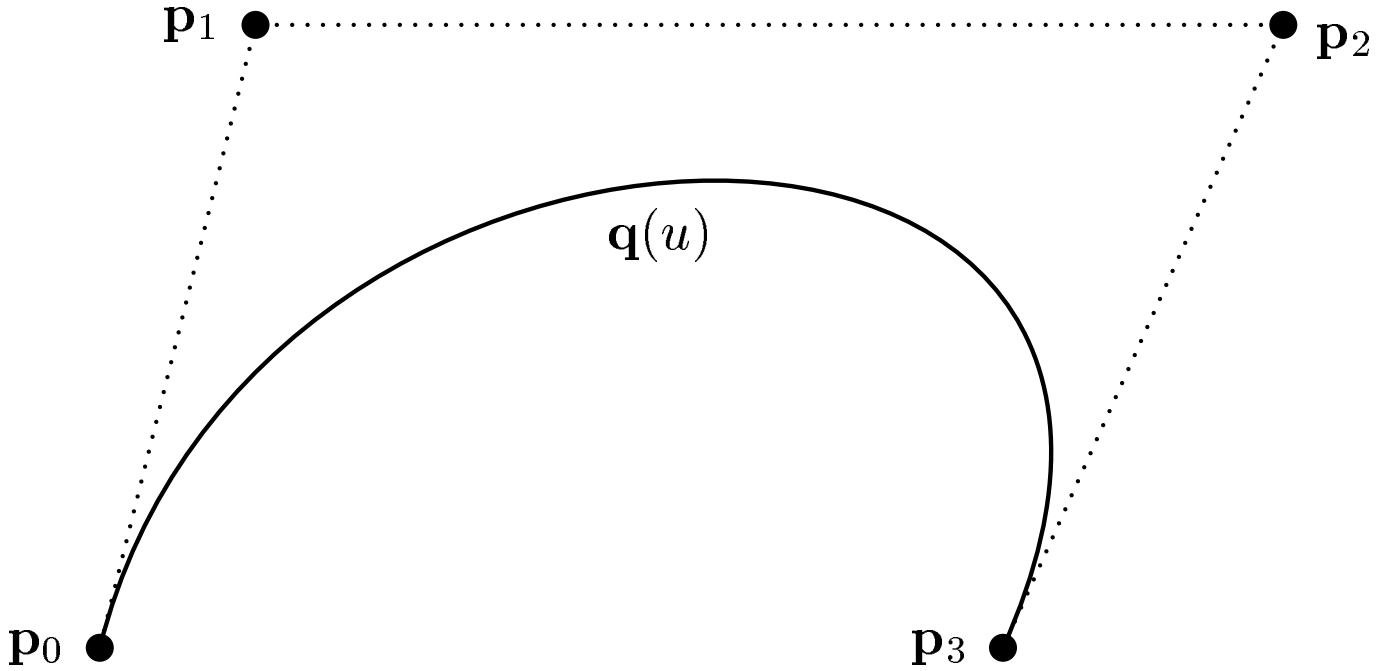
- po każdym `EmitVertex()` wartości zmiennych są nieokreślone

- za pomocą shadera geometrii można:
  - filtrować prymitywy
  - zmieniać typ prymitywu
  - rozdzielać prymitywy po warstwach tekstury sześcianowej (tworzyć kopie warstw)
  - obliczać parametry, dotyczące całego prymitywa (wektor normalny do ściany)
  - dla prymitywów z przyleganiem można obliczyć dla wierzchołków wektory styczne i normalne
  - utworzyć nową geometrię, nie jest zalecane tworzenie dużej ilości nowych wierzchołków
    - dla generacji dużych ilości nowej geometrii używa się shaderów tesselacji (tesselation control shader, tesselation evaluation shader), dostępnych od OpenGL 4.0

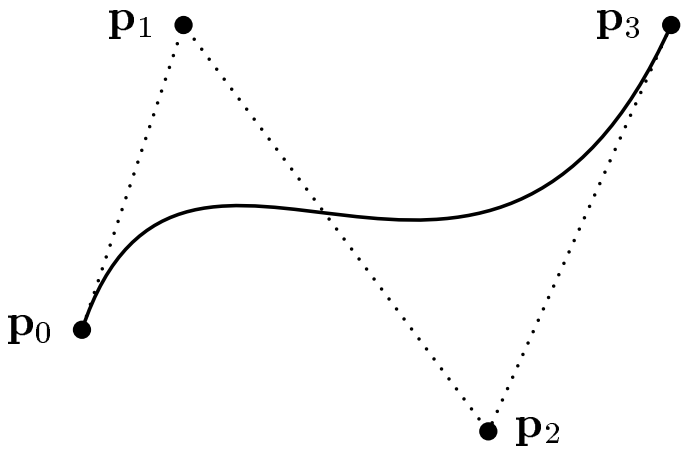
Krzywe Béziera

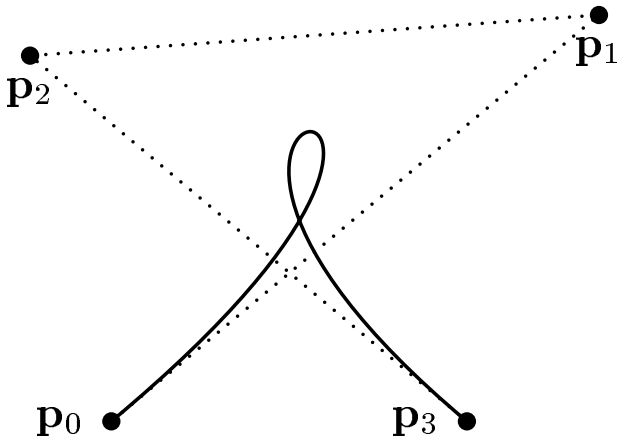
- Krzywe Béziera
  - Bézier — Renault, 1968, 1974;
  - de Casteljau — Citroën 1959, 1963
- B-splajny (Shoenberg 1946)

Krzywe Béziera trzeciego stopnia



Krzywe Béziera trzeciego stopnia

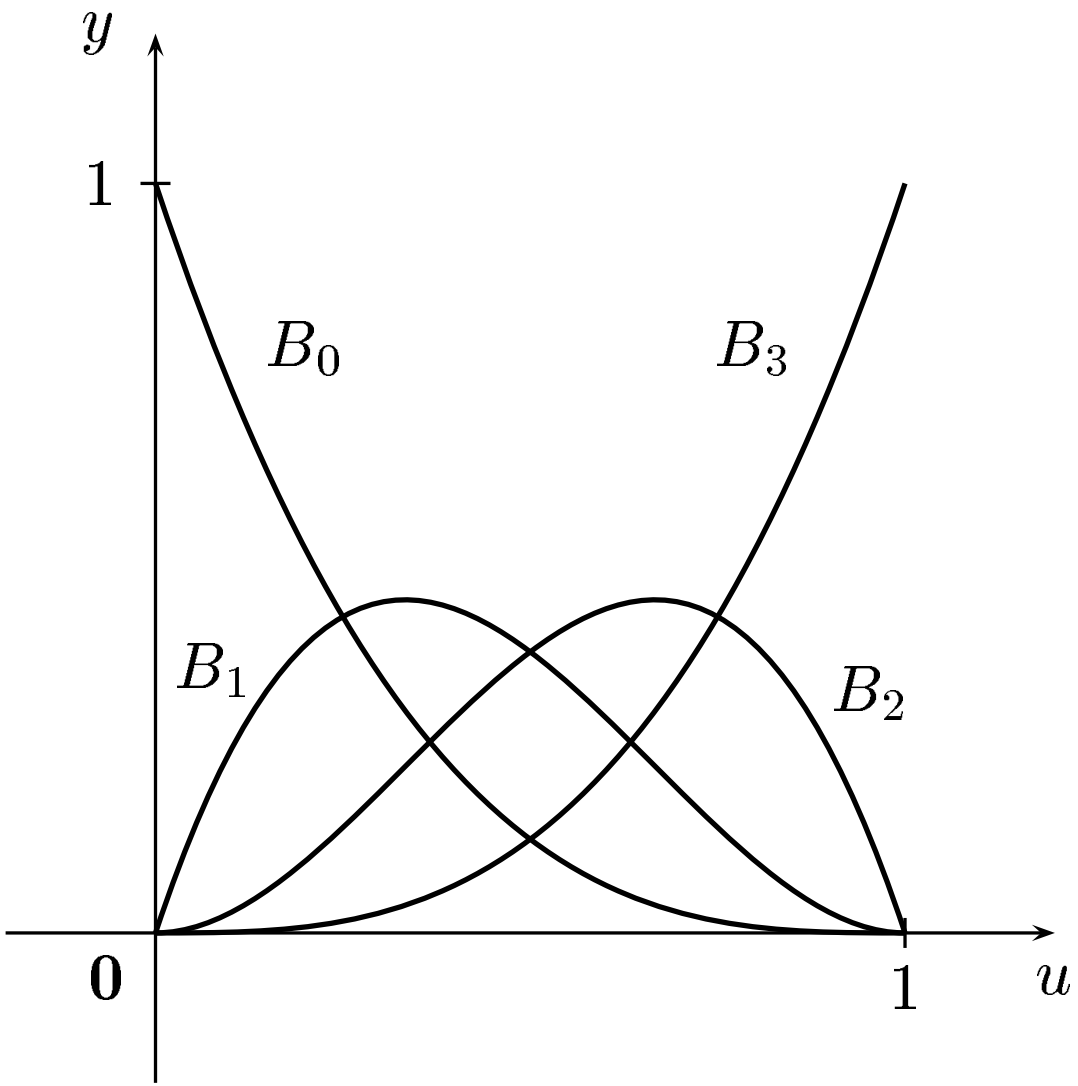




Krzywe Béziera trzeciego stopnia

- $q(u) = B_0(u)p_0 + B_1(u)p_1 + B_2(u)p_2 + B_3(u)p_3$, gdzie
  - $B_i(u) = \binom{3}{i}u^i(1-u)^{3-i}$ — wielomiany Bernsteina,
  - $\binom{n}{m} = C_n^m = \frac{n!}{m!(n-m)!}$ — symbol Newtona
  - $B_0(u) = (1-u)^3, \quad B_1(u) = 3u(1-u)^2$
  - $B_2(u) = 3u^2(1-u), \quad B_3(u) = u^3$
  - $\sum_{i=0}^{3} B_i(u) = \sum_{i=0}^{3} \binom{3}{i}u^i(a-u)^{3-i} = \left(u + (1-u)\right)^3 = 1$

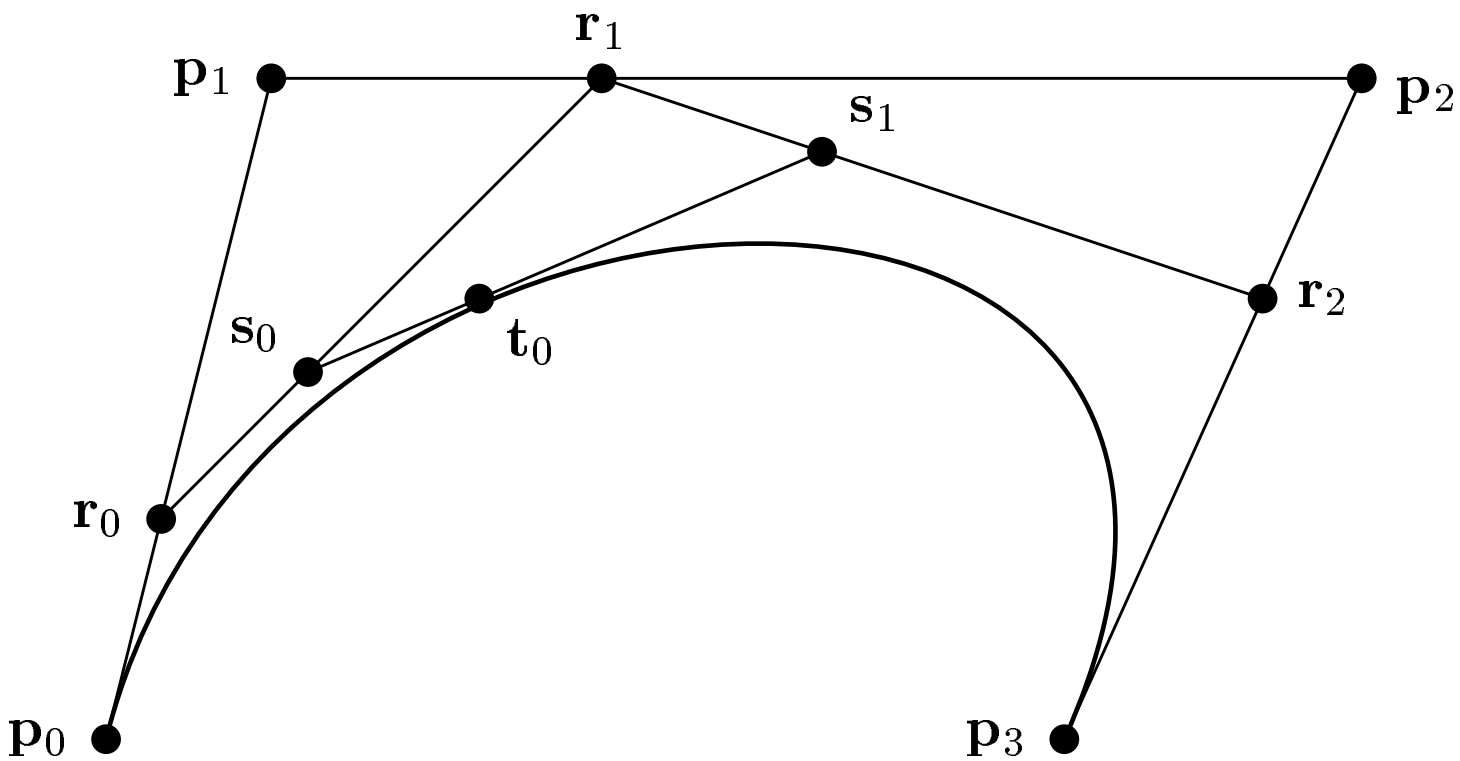
Wielomiany Bernsteina (stopnia 3)



Wielomiany Bernsteina (stopnia 3)

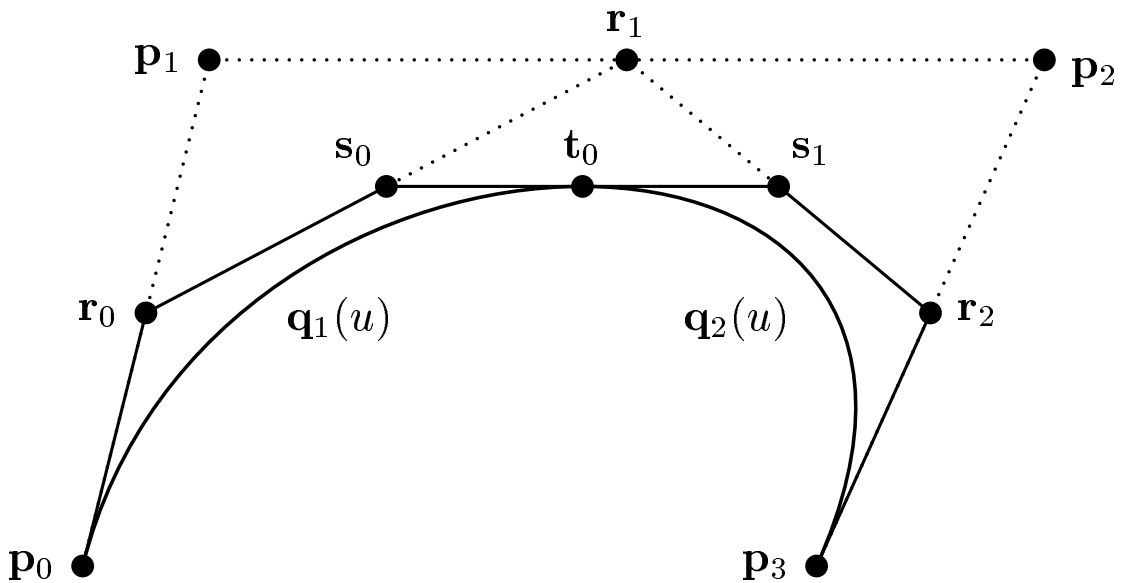
$$B_0'(0) = -3, \quad B_1'(0) = 3, \quad B_2'(0) = 0, \quad B_3'(0) = 0$$
$$B_0'(1) = 0, \quad B_1'(1) = 0, \quad B_2'(1) = -3, \quad B_3'(1) = 3$$

$$q'(0) = 3(p_1 - p_0),$$
$$q'(1) = 3(p_3 - p_2)$$



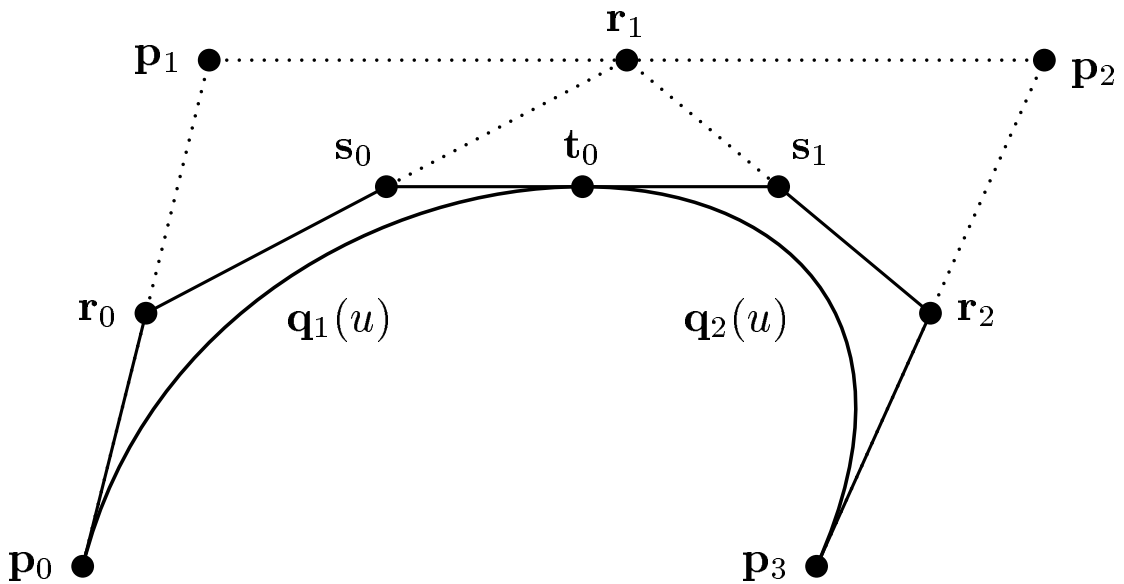
$$r_i = (1-u)\cdot p_i + u \cdot p_{i+1},$$
$$s_i = (1-u)\cdot r_i + u \cdot p_{i+1},$$
$$t_0 = (1-u)\cdot s_0 + u s_1$$

Algorytm de Casteljau ($u = \frac{1}{2}$)



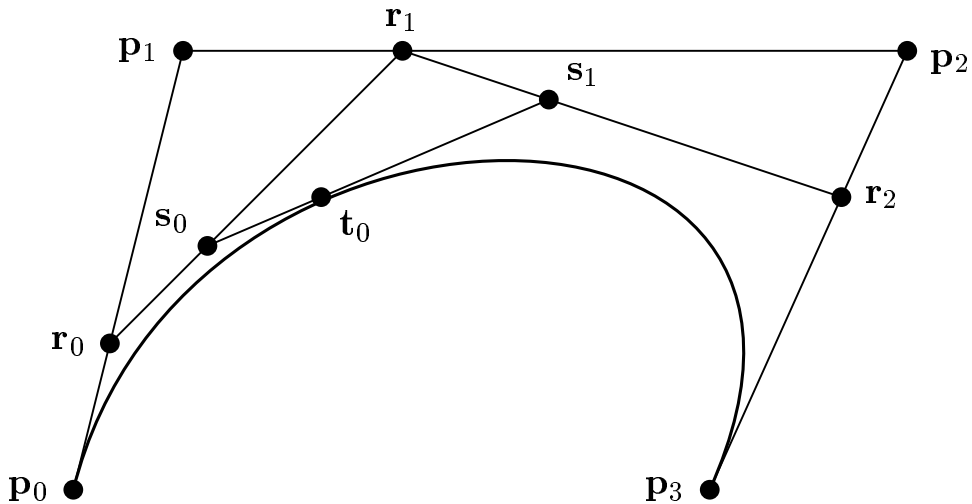
$$r_i = \frac{p_i + p_{i+1}}{2}, \quad s_i = \frac{r_i + r_{i+1}}{2}, \quad t_0 = \frac{s_0 + s_1}{2},$$

$$q(1/2) = t_0 = \frac{1}{8}p_0 + \frac{3}{8}p_1 + \frac{3}{8}p_2 + \frac{1}{8}p_3$$

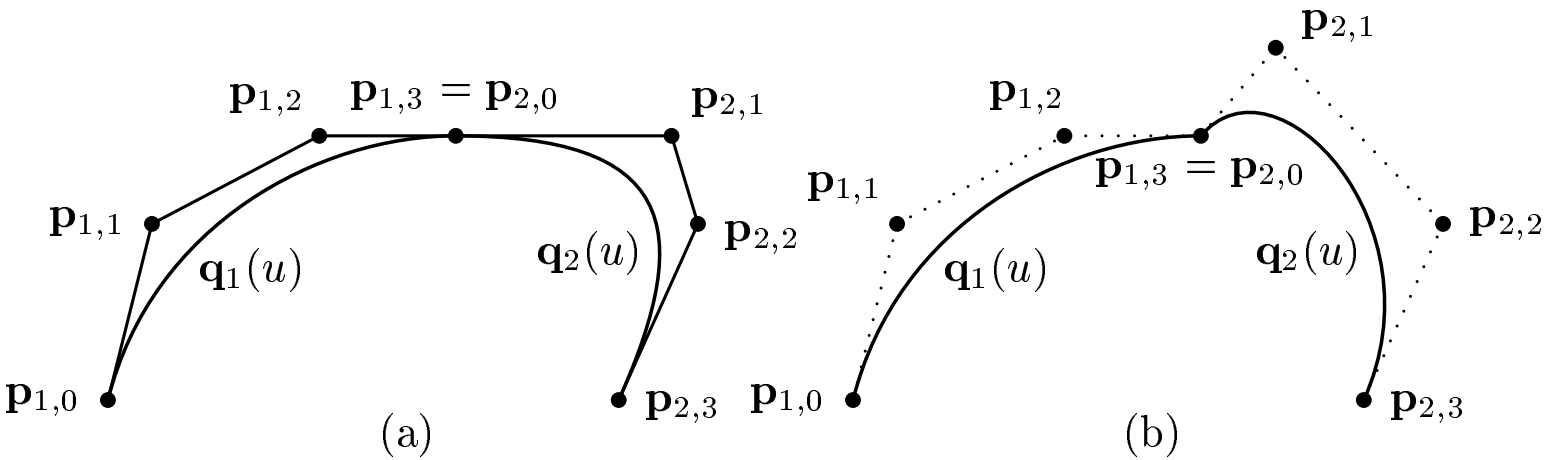


Twierdzenie 1. Niech $q(u)$ będzie krzywą Béziera o punktach kontrolnych p_0, p_1, p_2, p_3. Wtedy $q_1(u) = q(u/2)$ będzie Krzywą Béziera o punktach kontrolnych p_0, r_0, s_0, t_0,
$q_2(u) = q((u+1)/2)$ będzie krzywą Béziera o punktach t_0, s_1, r_2, p_3.

Zagęszczanie (recursive subdivision)



Twierdzenie 2. Niech $q(u)$ będzie krzywą Béziera o punktach kontrolnych p_0, p_1, p_2, p_3. Wtedy $q_1(u) = q(u_0 u)$ będzie Krzywą Béziera o punktach kontrolnych p_0, r_0, s_0, t_0,
$q_2(u) = q(u_0 + (1 - u_0)u)$ będzie krzywą Béziera o punktach t_0, s_1, r_2, p_3.



$$q_1'(1) = q_2'(0) \Rightarrow p_{1,3} - p_{1,2} = p_{2,1} - p_{2,0}$$

Zagadnienie interpolacji

- Dane są punkty $p_0, \ldots, p_m$ i węzły $u_0, \ldots, u_m$.
- Określić parametryzowaną krzywą $q(u)$ tak, żeby $q(u_i) = p_i$ dla $i = 0, \ldots, m$.
- Krzywa odcinkowo-wielomianowa (trzeciego stopnia).
- Sklejanie krzywych Béziera.

Splajny Catmulla-Roma

- Dane są punkty $P_0, \ldots, P_m$ i węzły $u_i = i$ dla $i = 0, \ldots, m$.
- Określić parametryzowaną krzywą $q(u)$ tak, żeby $q(i) = P_i$ dla $i = 1, \ldots, m-1$.
- Krzywa Catmull-Rom składa się z $m-2$ krzywych Béziera.
- Punkty kontrolne wybiera się tak, żeby krzywa była klasy C^1.

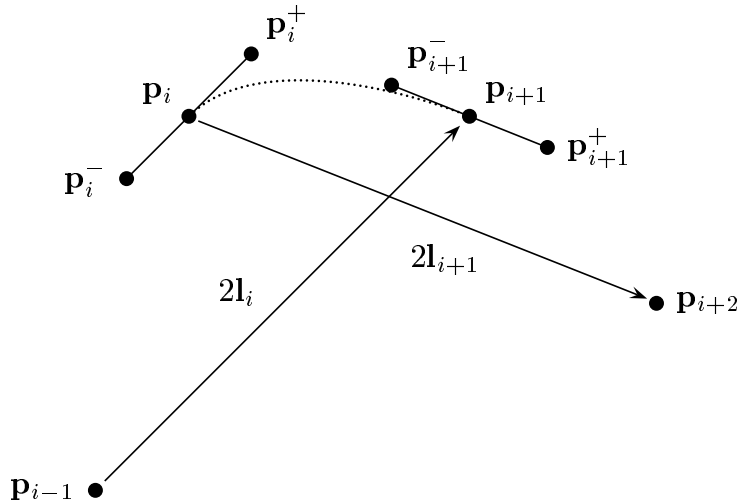


Figure VII.22: Defining the Catmull-Rom spline segment from the point $\mathbf{p}_i$ to the point $\mathbf{p}_{i+1}$. The points $\mathbf{p}_i^-$, $\mathbf{p}_i$, and $\mathbf{p}_i^+$ are collinear and parallel to $\mathbf{p}_{i+1} - \mathbf{p}_{i-1}$. The points $\mathbf{p}_i$, $\mathbf{p}_i^+$, $\mathbf{p}_{i+1}^-$, and $\mathbf{p}_{i+1}$ form the control points of a degree three Bézier curve, which is shown as a dotted curve.

$$l_i = \frac{1}{2}(p_{i+1} - p_{i-1}), \quad p_i^{\pm} = p_i \pm \frac{1}{3} l_i$$

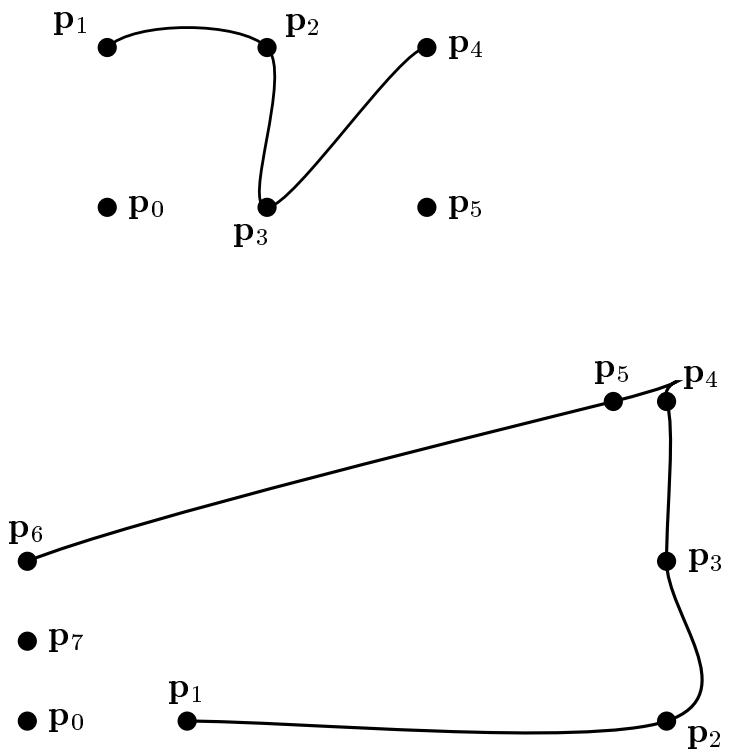
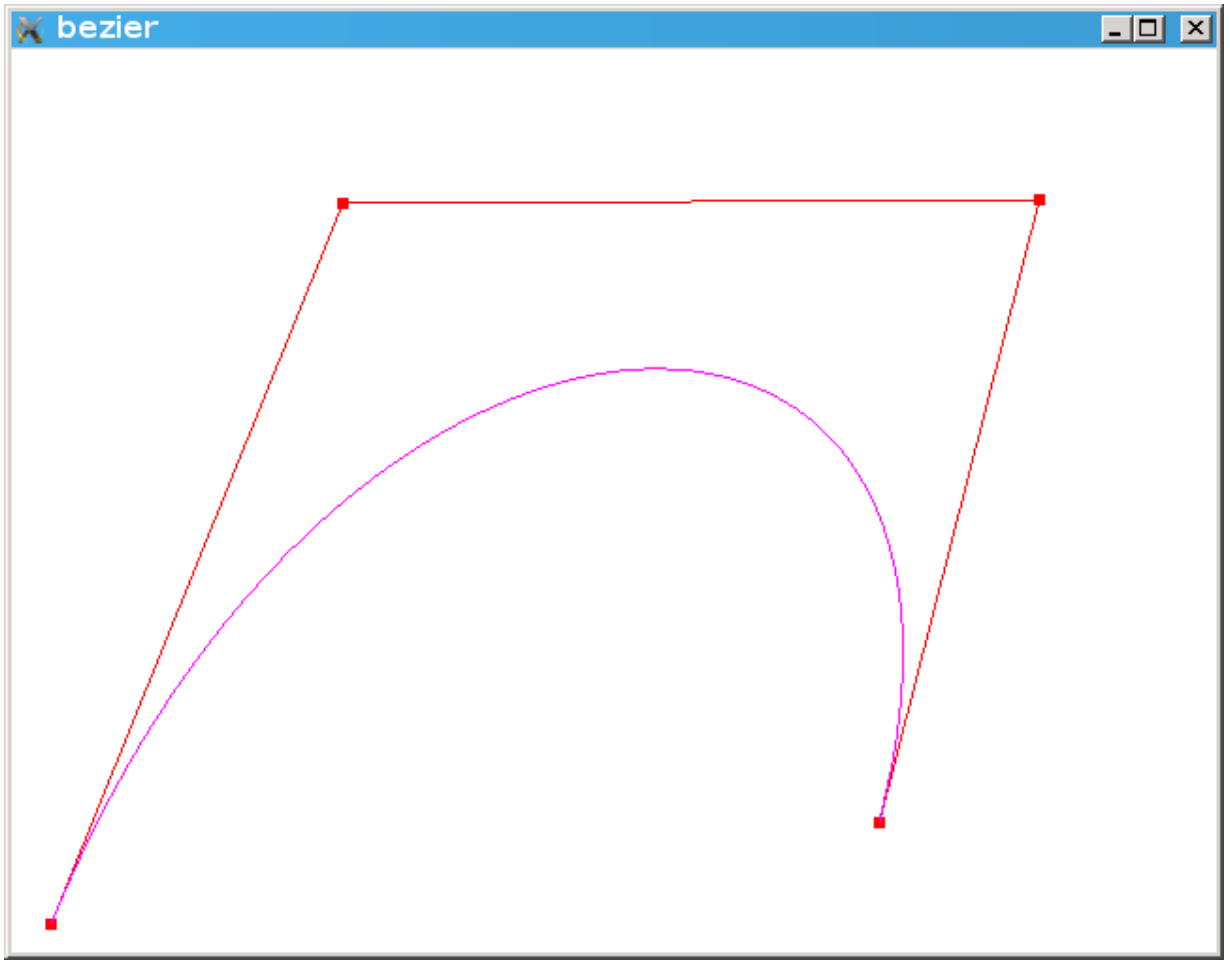


Figure VII.23: Two examples of Catmull-Rom splines with uniformly spaced knots.

Aplikacja

Przykładowy Render



- `Window` — odpowiada za kontekst
- `Bezier` — model: VAO, renderowanie
- `Program` — program: kompilacja shaderów
- `glerror.h` — definicje, związane z debugowaniem
- Kod przeważnie jest zgodny z Google C++ Style Guide

Założenie

- Obieky geometryczne oraz programy definijujemy jako zmienne (nie dynamicznr) klasy `Window`
- Kod, odpowiadający za zwolnienie zasobów karty graficznej umieszczemy w destruktorach klas-programów i klas-obiektów geometryczny
  - w taki sposób, zostaną one odpalone po zakończeniu pacy, w wyniku odpalania standardowego destruktora obiektu klasy `Window`
- Inicjalizację obiektów geometrycznych i programów umieszczamy w funkcjach `Initialize` odpowiednich klas

```
#version 430

layout(location=0) in vec2 in_position;

out vec4 frag_color;

void main(void){
    gl_Position = vec4(in_position,0,1);
    frag_color = vec4(1,0,0,1);
}
```

```
#version 430

layout (lines_adjacency) in;
layout (line_strip, max_vertices=65) out;

out vec4 frag_color;

void main(void){
    int i;
    float t=0.0,h=1.0/64.0;

    frag_color=vec4(1,0,1,1);

    vec4 q0, q1, q2, r0, r1, s;
    gl_Position = gl_in[0].gl_Position;
    EmitVertex();
 ..............................
```

Shader Geometrii. Pętla i zakończenie


```
.............................
  for(i=1; i<64; i++){
    t +=h;
    q0=mix(gl_in[0].gl_Position,
           gl_in[1].gl_Position,t);
    q1=mix(gl_in[1].gl_Position,
           gl_in[2].gl_Position,t);
    q2=mix(gl_in[2].gl_Position,
           gl_in[3].gl_Position,t);
    r0=mix(q0, q1,t);
    r1=mix(q1, q2,t);
    gl_Position = mix(r0, r1, t);
    EmitVertex();
  }
  gl_Position = gl_in[3].gl_Position;
  EmitVertex();
  EndPrimitive();
}
```

```
#version 430
layout (location = 0) out vec4 color;

in vec4 frag_color;

void main(void){
    color = frag_color;
}
```

Globalny obiekt window i callbacki


```
const int kMajorGLVersion = 4;
const int kMinorGLVersion = 3;
Window window("Krzywe Beziera", 800, 600);
void Resize (int new_width, int new_height){
    window.Resize(new_width, new_height);
}
void Render(){
    window.Render();
}
void KeyPressed(unsigned char key,
                int x_coord, int y_coord){
    window.KeyPressed(key, x_coord, y_coord);
}
void MouseButton(int button, int state,
                 int x_coord, int y_coord){
    window.MouseButton(button, state, x_coord, y_coord)
}
```

```
int main(int argc, char* argv[]){
    window.Initialize(argc, argv,
          kMajorGLVersion, kMinorGLVersion);

    glutReshapeFunc(Resize);
    glutDisplayFunc(Render);
    glutKeyboardFunc(KeyPressed);
    glutMouseFunc(MouseButton);

    glutMainLoop();

    exit(EXIT_SUCCESS);
}
```

Klasa Window, dane prywatne


```
int width_;
int height_;
char title_[256];
Bezier bezier_;
Program program_;
Program bezier_program_;

void InitGlutOrDie(int argc, char* argv[],
    int major_gl_version, int minor_gl_version);
void InitGlewOrDie();
void InitModels();
void InitPrograms();
```

Klasa Window. Dane publiczne


```
Window(const char*, int, int);
void Initialize(int argc, char* argv[],
  int major_gl_version, int minor_gl_version);
void Resize(int new_width, int new_height);
void Render(void);
void KeyPressed(unsigned char key,
                int x_coord, int y_coord);
void MouseButton(int button, int state,
                int x_coord, int y_coord);
```

Trzy stałe i konstruktor


```
const char* kVertexShader = "Shader.vertex.glsl";
const char* kGeometryShader = "Shader.geometry.glsl";
const char* kFragmentShader = "Shader.fragment.glsl";


Window::Window(const char * title,
               int width, int height){
    strcpy(title_, title);
    width_ = width;
    height_ = height;
}
```

```
void Window::Initialize(int argc, char * argv[],
    int major_gl_version, int minor_gl_version){
  InitGlutOrDie(argc, argv,
      major_gl_version, minor_gl_version);
  InitGlewOrDie();
  std::cout << "OpenGL initialized: OpenGL version: "
            << glGetString(GL_VERSION)
            << " GLSL version: "
            << glGetString(GL_SHADING_LANGUAGE_VERSION)
            << std::endl;
  InitModels();
  InitPrograms();
  glEnable(GL_PROGRAM_POINT_SIZE);
  glPointSize(6);
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
}
```

Inicjalizacja kontekstu


```
void Window::InitGlutOrDie(int argc, char * argv[],
    int major_gl_version, int minor_gl_version){
 glutInit(&argc, argv);
 glutInitContextVersion(major_gl_version,
                        minor_gl_version);
 glutInitContextProfile(GLUT_CORE_PROFILE);
 glutSetOption(
    GLUT_ACTION_ON_WINDOW_CLOSE,
    GLUT_ACTION_GLUTMAINLOOP_RETURNS);
 glutInitWindowSize(width_, height_);
 glutInitDisplayMode(GLUT_DOUBLE
              /* | GLUT_DEPTH */ | GLUT_RGBA);
 int window_handle = glutCreateWindow(title_);
 if( window_handle < 1) {
   std::cerr << "ERROR: " << std::endl;
   exit(EXIT_FAILURE);
  }
}
```

Inicjalizacja kontekstu. Profil DEBUG


```
 ..............................
 glutInitContextProfile(GLUT_CORE_PROFILE);
#ifdef DEBUG
 glutInitContextFlags(GLUT_DEBUG);
#endif
 glutSetOption(

 ..............................
```

```
void Window::InitGlewOrDie(){
  GLenum glew_init_result;
  glewExperimental = GL_TRUE;
  glew_init_result = glewInit();

  if (GLEW_OK != glew_init_result) {
    std::cerr << "Glew ERROR: "
      << glewGetErrorString(glew_init_result)
      << std::endl;
    exit(EXIT_FAILURE);
  }
}
```

Profil DEBUG — rejesrtacja callbacka na error


```
.............................
#ifdef DEBUG
 if(glDebugMessageCallback){
   std::cout << "Register OpenGL debug callback ";
   glEnable(GL_DEBUG_OUTPUT_SYNCHRONOUS);
   glDebugMessageCallback(OpenglCallbackFunction, NULL)
   GLuint unused_ids = 0;
   glDebugMessageControl(GL_DONT_CARE,
     GL_DONT_CARE,
     GL_DONT_CARE,
     0,
     &unused_ids,
     GL_FALSE);
 }
 else
   std::cout << "glDebugMessageCallback not available";
#endif
}
```

Inicjalizacja modelu i programu


```
void Window::InitModels(){
    bezier_.Initialize();
}

void Window::InitPrograms(){
    program_.Initialize(kVertexShader,
          kFragmentShader);
    bezier_program_.Initialize(kVertexShader,
          kFragmentShader, kGeometryShader);
}
```

```
void Window::Resize(int new_width, int new_height){
    width_ = new_width;
    height_ = new_height;
    glViewport(0, 0, width_, height_);
    glutPostRedisplay();
}
```

```
void Window::KeyPressed(unsigned char key,
     int /*x_coord*/, int /*y_coord*/){
  switch (key){
    case 27:
      glutLeaveMainLoop();
    break;
    default:
    break;
  }
}
```

```
void Window::MouseButton(int button, int state,
                         int x_coord, int y_coord){
  if (button == GLUT_LEFT_BUTTON) {
    // when the button is released
    if (state == GLUT_UP) {
      if(bezier_.AddPoint(
        (float)x_coord*2.0f/width_ - 1.0f,
        -(float)y_coord*2.0f/height_ + 1.0f))
      {
          glutPostRedisplay();
      }
    }
  }
}
```

```
void Window::Render(){
    glClear(GL_COLOR_BUFFER_BIT);

    bezier_.Draw(program_, bezier_program_);

    glutSwapBuffers();
}
```

```
class Program{
public:
    void Initialize(const char* vertex_shader_file,
        const char* fragment_shader_file,
        const char* geometry_shader_file=NULL);
    operator GLuint() const{return program_;}
    ~Program();
private:
    GLuint program_;
    GLuint vertex_shader_;
    GLuint fragment_shader_;
    GLuint geometry_shader_;
    GLuint LoadAndCompileShaderOrDie(
      const char* source_file, GLenum type);
    GLuint LinkProgramOrDie(GLint vertex_shader,
      GLint fragment_shader,
      GLint geometry_shader=0);
};
```

Inicjalizacja. Kompilacja shaderów


```
void Program::Initialize(
            const char* vertex_shader_file,
            const char* fragment_shader_file,
            const char* geometry_shader_file){
  vertex_shader_ = LoadAndCompileShaderOrDie(
    vertex_shader_file, GL_VERTEX_SHADER);
  fragment_shader_ = LoadAndCompileShaderOrDie(
    fragment_shader_file, GL_FRAGMENT_SHADER);

    if(geometry_shader_file == NULL){
        geometry_shader_ = 0;
    }
    else{
        geometry_shader_ = LoadAndCompileShaderOrDie(
          geometry_shader_file, GL_GEOMETRY_SHADER);
    }
}
```

```
    program_ = LinkProgramOrDie(vertex_shader_,
                                fragment_shader_,
                                geometry_shader_);
    glUseProgram(program_);
    // some actions with a new program
    glUseProgram(0);
}
```

```
GLuint Program::LoadAndCompileShaderOrDie(
  const char * source_file, GLenum type){
  int file_size;
  char * shader_code;
  GLuint shader=glCreateShader(type);
  ifstream file (source_file,
                 ios::in|ios::ate|ios::binary);
  if (file.is_open()) {
    file_size = file.tellg();
    shader_code = new char [file_size+1];
    file.seekg (0, ios::beg);
    file.read (shader_code, file_size);
    shader_code[file_size]='\0';
    file.close();
    glShaderSource(shader, 1,
           (const GLchar**) &shader_code, NULL);
    glCompileShader(shader);
    delete[] shader_code;
```

Kompilacja shadera, niepowodzenie otwarcia pliku


```
} else { //file was not opened
    cerr << "Could not open the file "
         << source_file
         << endl;
    exit( EXIT_FAILURE );
}
```

Kompilacja shadera. Weryfikacja wyniku


```
GLint  compiled;
glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
if (!compiled) {
  switch(type){
  case GL_VERTEX_SHADER:
      cerr << "vertex ";
  break;
  case GL_FRAGMENT_SHADER:
      cerr << "fragment ";
  break;
  case GL_GEOMETRY_SHADER:
      cerr << "geometry ";
  break;
  }
```

Kompilacja shadera. Weryfikacja wyniku, cd


```
    GLint  log_size;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH,&log_size)
    char* log_msg = new char[log_size];
    glGetShaderInfoLog(shader, log_size, NULL, log_msg)
    cerr << log_msg << endl;
    delete [] log_msg;
    exit( EXIT_FAILURE );
  }
  return shader;
}
```

Linkowanie programu.


```
GLuint Program::LinkProgramOrDie(
        GLint vertex_shader,
        GLint fragment_shader,
        GLint geometry_shader){

    GLuint new_program = glCreateProgram();
    glAttachShader(new_program, vertex_shader);
    if(geometry_shader>0){
        glAttachShader(new_program, geometry_shader);
    }
    glAttachShader(new_program, fragment_shader);
    glLinkProgram(new_program);
.............................
```

Linkowanie programu. Weryfikacja wyniku


```
  GLint  linked;
  glGetProgramiv(new_program, GL_LINK_STATUS, &linked);
  if ( !linked ) {
    std::cerr << "Shader program failed to link";
    GLint  log_size;
    glGetProgramiv(new_program, GL_INFO_LOG_LENGTH,
                   &log_size);
    char* log_msg = new char[log_size];
    glGetProgramInfoLog(new_program, log_size, NULL,
                        log_msg);
    std::cerr << log_msg << std::endl;
    delete [] log_msg;
    exit( EXIT_FAILURE );
  }
  return new_program;
}
```

Usuwanie programu z pamięci GPU


```
Program::~Program(){
    glUseProgram(0);
    glDetachShader(program_, vertex_shader_);
    glDetachShader(program_, fragment_shader_);
    if(geometry_shader_>0){
        glDetachShader(program_, geometry_shader_);
        glDeleteShader(geometry_shader_);
    }

    glDeleteShader(fragment_shader_);
    glDeleteShader(vertex_shader_);
    glDeleteProgram(program_);
}
```

```
class Bezier{
 public:
    void Initialize();
    void Draw(const Program & program,
              const Program & bezier_program);
    ~Bezier();
    Bezier(void);
    bool AddPoint(float x_coord, float y_coord);
 private:
    GLuint vao_;
    GLuint vertex_buffer_;

    unsigned int max_points_;
    unsigned num_points_;
}
```

```
Bezier::Bezier(void){
    max_points_=4;
    num_points_=0;
}
```

```
void Bezier::Initialize(){
  glGenVertexArrays(1, &vao_);
  glBindVertexArray(vao_);

  glGenBuffers(1, &vertex_buffer_);
  glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_);
  glBufferData(GL_ARRAY_BUFFER,
      max_points_*2*sizeof(float), NULL,
      GL_DYNAMIC_DRAW);
  glVertexAttribPointer(0, 2, GL_FLOAT,
      GL_FALSE, 0, (GLvoid*)0);
  glEnableVertexAttribArray(0);

  glBindVertexArray(0);
}
```

```
void Bezier::Draw(const Program &program,
                  const Program &bezier_program){
    glBindVertexArray(vao_);
    glUseProgram(program);
    glDrawArrays(GL_LINE_STRIP, 0, num_points_);
    glDrawArrays(GL_POINTS, 0, num_points_);
    if(num_points_ == max_points_){
        glUseProgram(bezier_program);
        glDrawArrays(GL_LINES_ADJACENCY,
                     0, num_points_);
    }
    glBindVertexArray(0);
    glUseProgram(0);
}
```

Usuwanie z pamięci GPU


```
Bezier::~Bezier(){
    glDisableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glDeleteBuffers(1, &vertex_buffer_);

    glBindVertexArray(0);
    glDeleteVertexArrays(1,&vao_);

}
```

Dodanie punktu


```
bool Bezier::AddPoint(float x_coord, float y_coord){
  if(num_points_ == max_points_) return false;
  float vertex[2];
  vertex[0]=x_coord;
  vertex[1]=y_coord;
  glBindVertexArray(vao_);
  glBindBuffer(GL_ARRAY_BUFFER, vertex_buffer_);
  glBufferSubData(GL_ARRAY_BUFFER,
          (num_points_++)*2*sizeof(float),
          2*sizeof(float), vertex);
  glBindVertexArray(0);
  return true;
}
```