VGP352 – Week 1

- Agenda:
  - Course Intro
  - Reading technical papers
  - Curves
  - Curved surfaces
  - Per-fragment lighting revisited
    - Phong Shading
    - Surface-space
  - Bump mapping
    - Basic usage
    - Bumpmap storage

What should you already know?

- C++ and object oriented programming
  - For most assignments you will need to implement classes or portions of classes that conform to specific interfaces
- Graphics terminology and concepts
  - Polygon, pixel, texture, infinite light, point light, spot light, etc.
- Linear algebra and vector math
  - Matrix arithmetic

What should you already know?

- Material from VGP351:
  - Using OpenGL
    - Setting up shaders
    - Getting data in
    - etc.
  - Transformations
    - 3D space transformations
    - Projections
  - Lighting and shading
  - Texture mapping

What will you learn?

- Advanced lighting models
  - BRDFs
  - Fur and hair rendering
  - “Toon” and other non-photorealistic rendering

How will you be graded?

- Four bi-weekly quizzes
  - These are listed on the syllabus
- One final exam
- Three programming projects
  - The first will be pretty small...perhaps small enough to complete in class
  - The remaining two projects will be larger
- One in-class presentation

How will you be graded?

- Keep in mind:
  - There is a *lot more* reading than in VGP351
    - More readings from the textbook
    - Readings from academic papers
  - There is *more* programming than in VGP351

How will programs be graded?

- Does the program produce the correct output?
- Are appropriate algorithms and data-structures used?
- Is the code readable, clear, and properly documented?

How will the presentation be graded?

- During the term, several papers will be assigned to be read
  - Select and present one of the assigned readings to the class
  - Material from some papers may appear on bi-weekly quizzes

Class Web Site

- Syllabus, assignments, and base code:

http://people.freedesktop.org/~idr/2010Q4-VGP352/

Reading Technical Papers

- Why read technical papers?

Reading Technical Papers

- Why read technical papers?
  - Almost every major advance in gaming graphics can be traced to a SIGGRAPH paper from 1 to 10 years before
  - At publication many algorithms are not *yet* usable
    - Hardware isn't quite fast enough / flexible enough
    - Algorithm makes simplifying assumptions that prevent *general* use
- Reading papers *effectively* is a skill

Reading Technical Papers

- Read a paper in 3 passes:
  - 1st pass: the 10 minute overview
    - Read the title, abstract, and introduction
    - Read the section and subsection headings
    - Read the conclusion
    - Scan the references

Reading Technical Papers

- Read a paper in 3 passes:
  - 1st pass: the 10 minute overview
    - Read the title, abstract, and introduction
    - Read the section and subsection headings
    - Read the conclusion
    - Scan the references
  - Answer the “five C's”
    - Category: What type of paper is it?
    - Context: What other work is it related to?
    - Correctness: Is it based on valid / reasonable assumptions?
    - Contribution: What are the main contributions?
    - Clarity: Is it well written?

Reading Technical Papers

- Read a paper in 3 passes:
  - 2nd pass: an hour for details
    - Read the whole paper
    - Carefully examine diagrams, figures, graphs, etc.
    - Skip or skim proofs and detailed equations

Reading Technical Papers

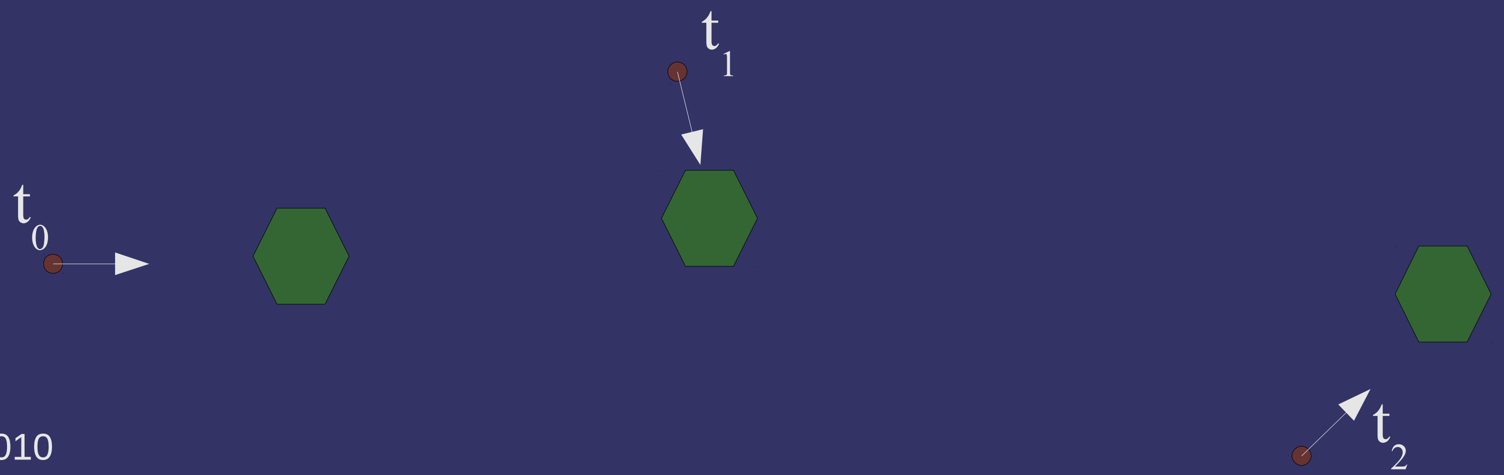
- Read a paper in 3 passes:
  - 3rd pass: re-implement the paper
    - Recreate the work
    - Identify and challenge every assumption in the paper
      - Helps identify *both* the innovations and failings of the paper
    - Compare you recreation with the original
    - Make notes of ideas for future work

References

Keshav, S. 2007. How to read a paper. *SIGCOMM Computer Communications Review.* 37, 3 (Jul. 2007), 83-84. http://www.sigcomm.org/ccr/drupal/files/p83-keshavA.pdf

Camera Control

- How can we move a virtual camera through a series of artist selected positions?



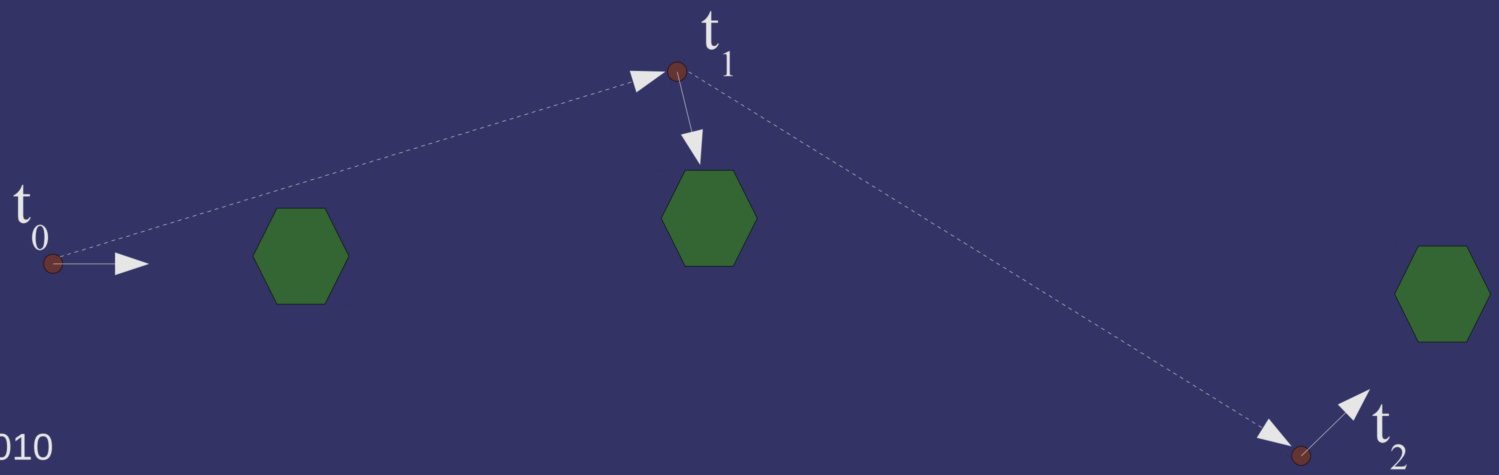
6-October-2010

Camera Control

- How can we move a virtual camera through a series of artist selected positions?
  - Linearly interpolate between the positions

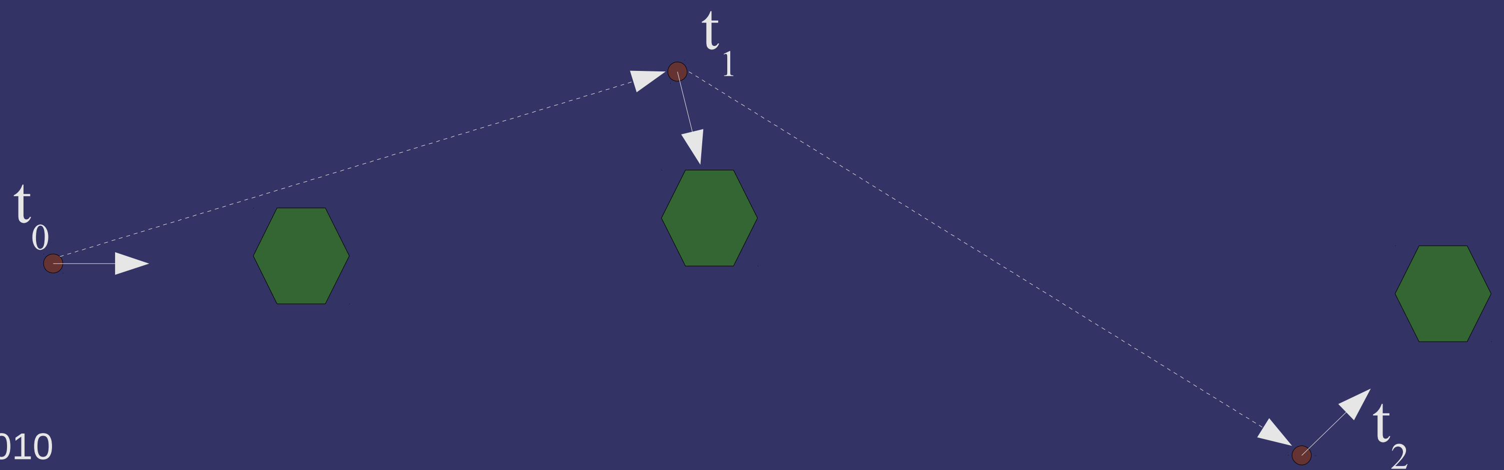
$$
\begin{aligned}
\mathbf{p}(t) &= \mathbf{p}_0 + t(\mathbf{p}_1 - \mathbf{p}_0) \\
&= (1-t)\mathbf{p}_0 + t\,\mathbf{p}_1
\end{aligned}
$$

  - Positionally continuous (aka *C^0 continuity*)



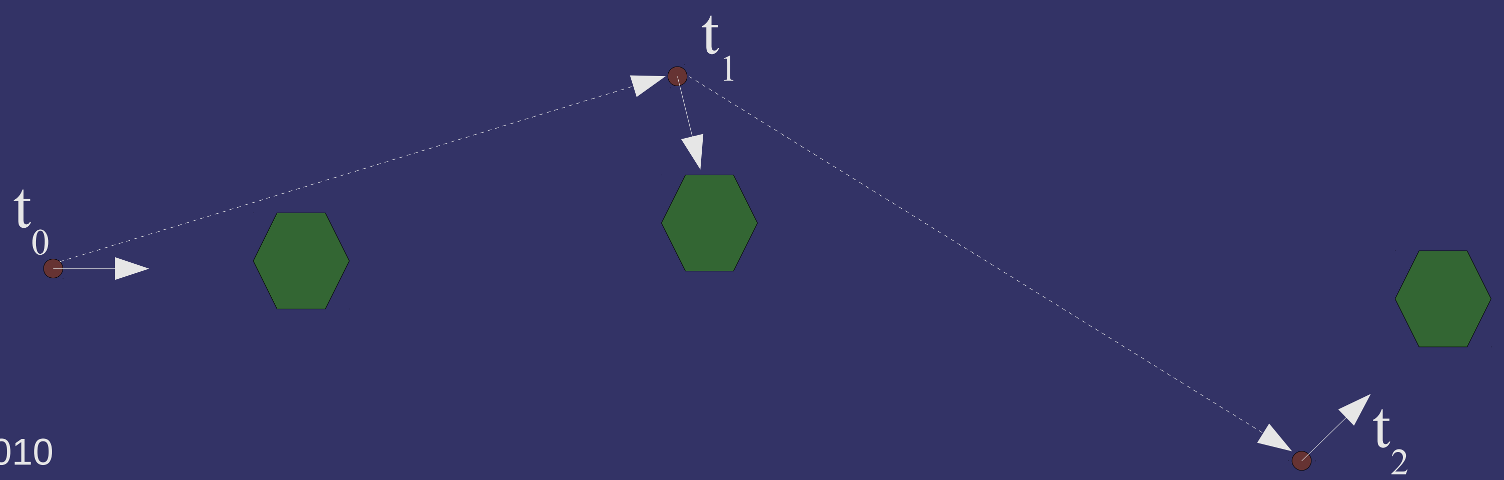
Camera Control

- What's wrong with C^0?



Camera Control

- What's wrong with C^0?
  - Jarring change in direction at control points
  - Jarring change in speed at control points
    - Direction change or speed change = velocity change

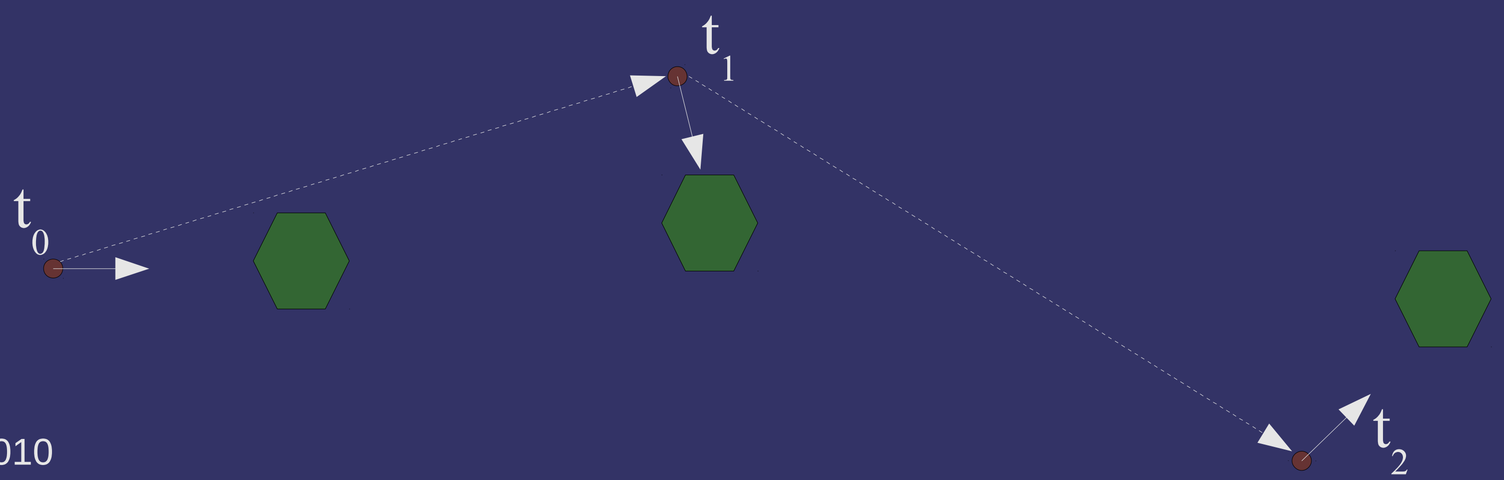


Continuity

- What does it mean?
  - "f is C^0" → The function f is continuous
  - "f is C^1" → The function f' is continuous
  - "f is C^2" → The function f'' is continuous
  - …
  - "f is C^∞" → All derivatives of f are continuous

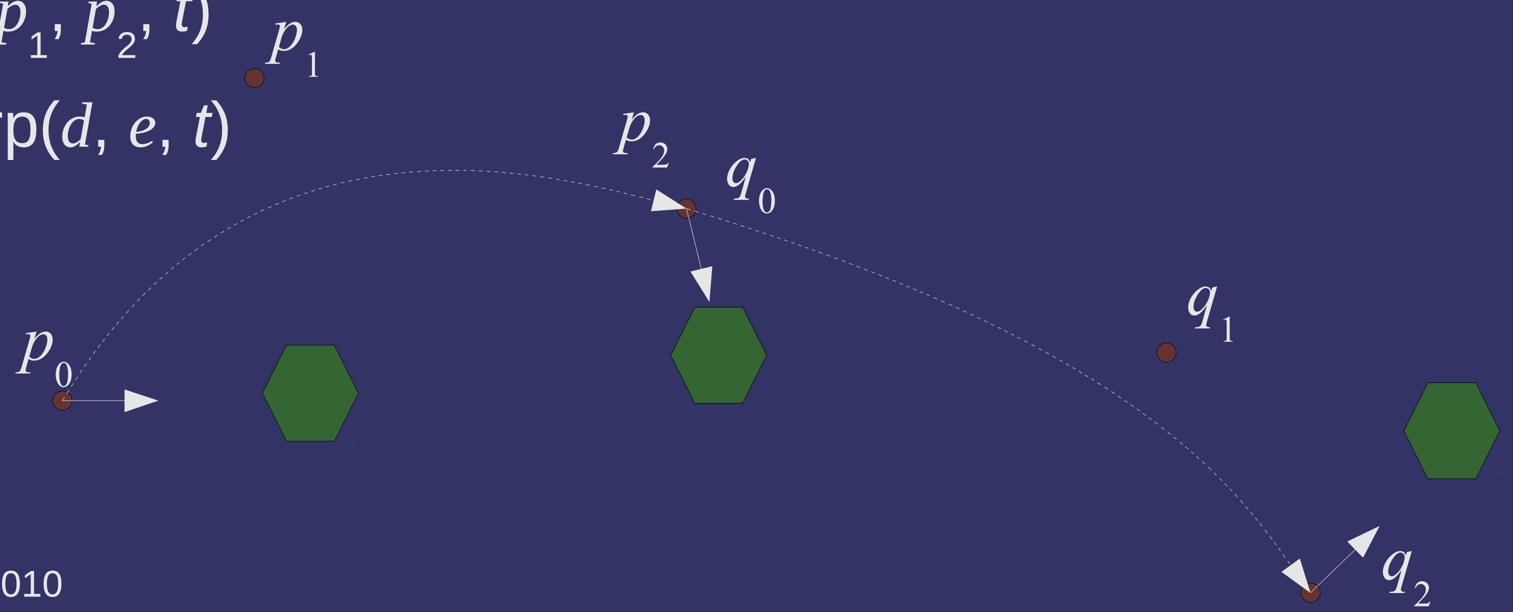
Camera Control

- How can we fix this?



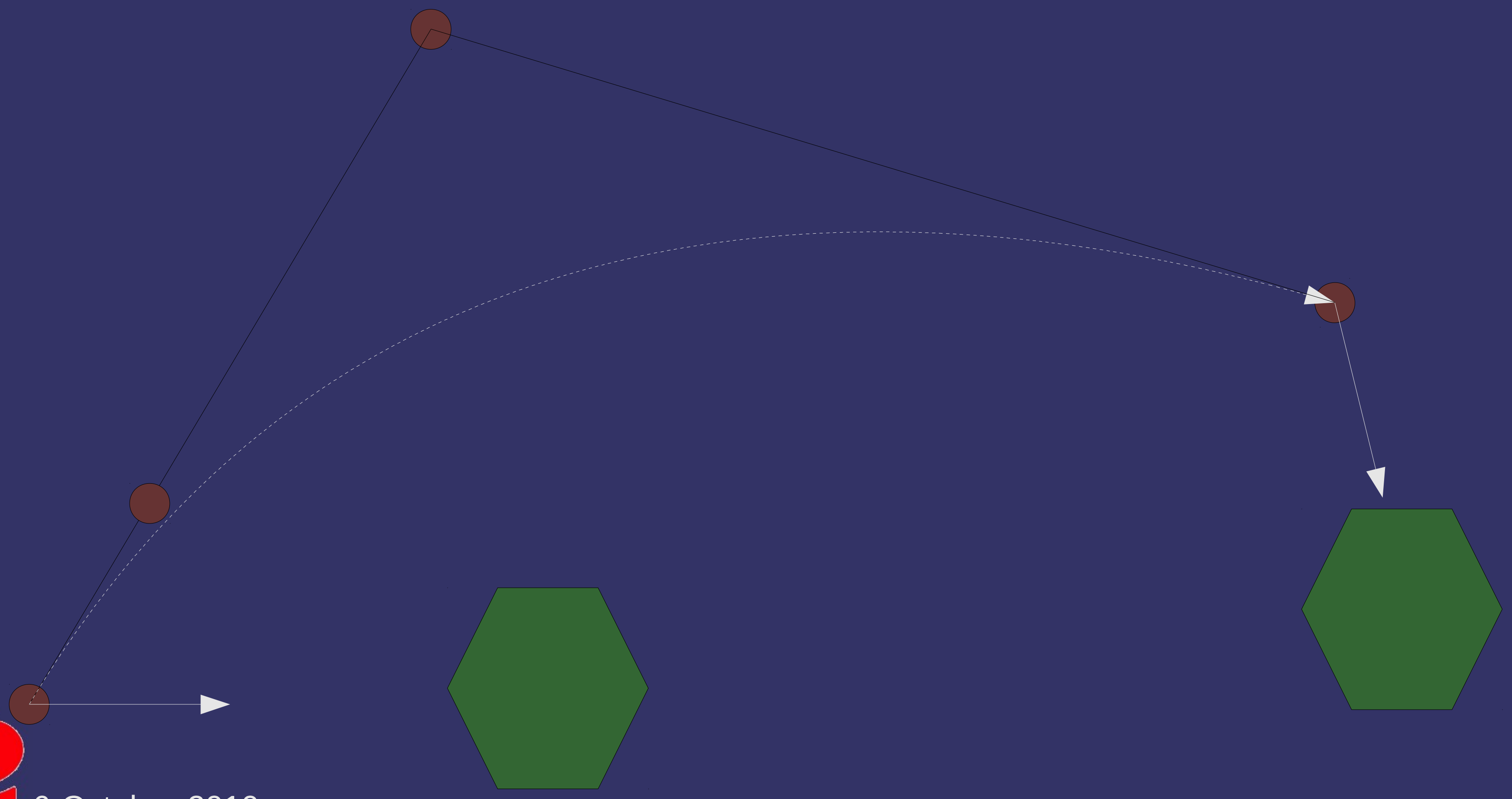
Camera Control

- How can we fix this?
  - Add one more control point for each segment
    - Each segment has p_0, p_1, and p_2
  - Do more linear interpolation
    - $d = \mathrm{lerp}(p_0, p_1, t)$
    - $e = \mathrm{lerp}(p_1, p_2, t)$
    - $\mathrm{p}(t) = \mathrm{lerp}(d, e, t)$



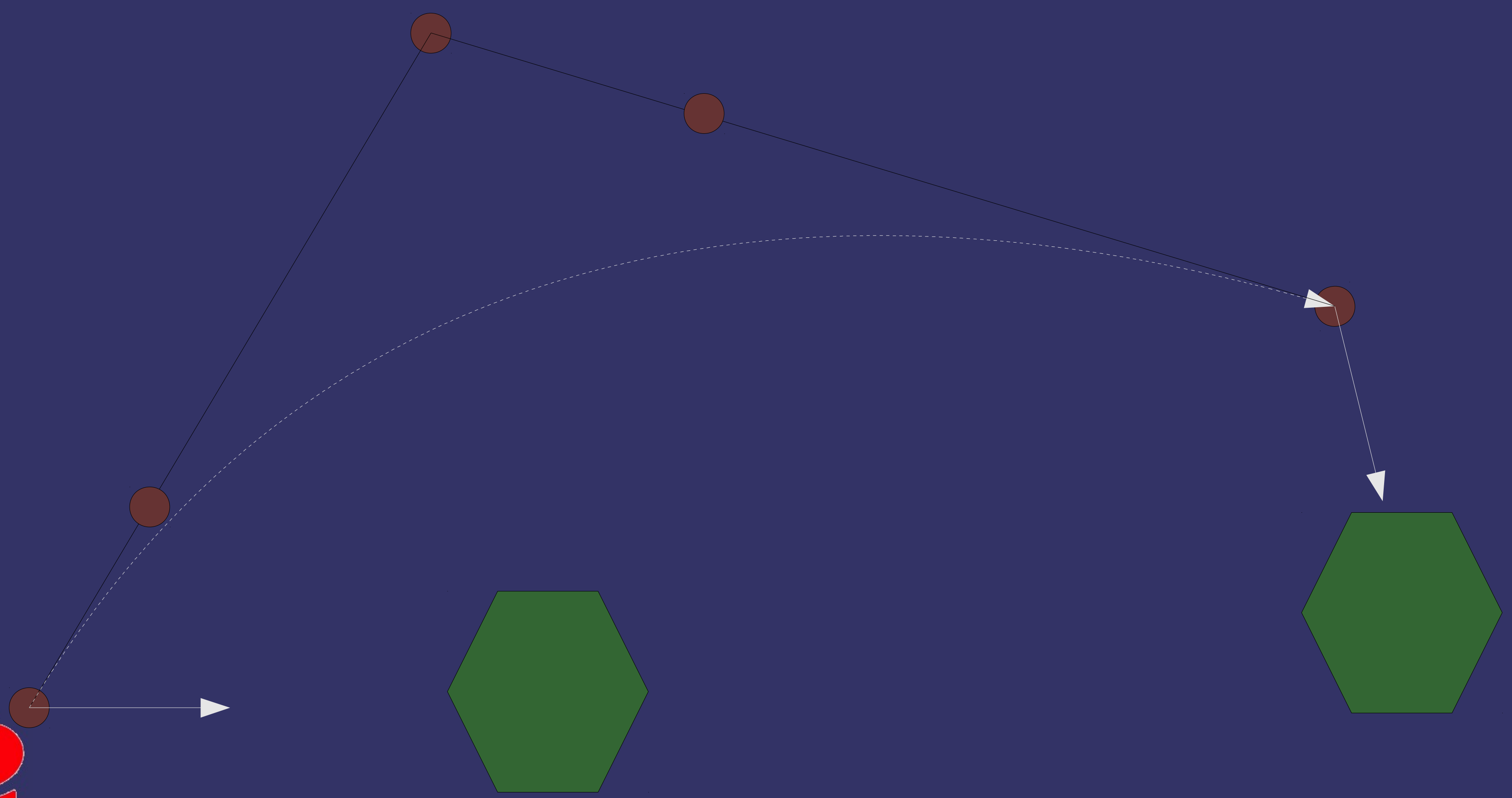
Camera Control

- p(0.3) = lerp(p_0, p_1, 0.3)



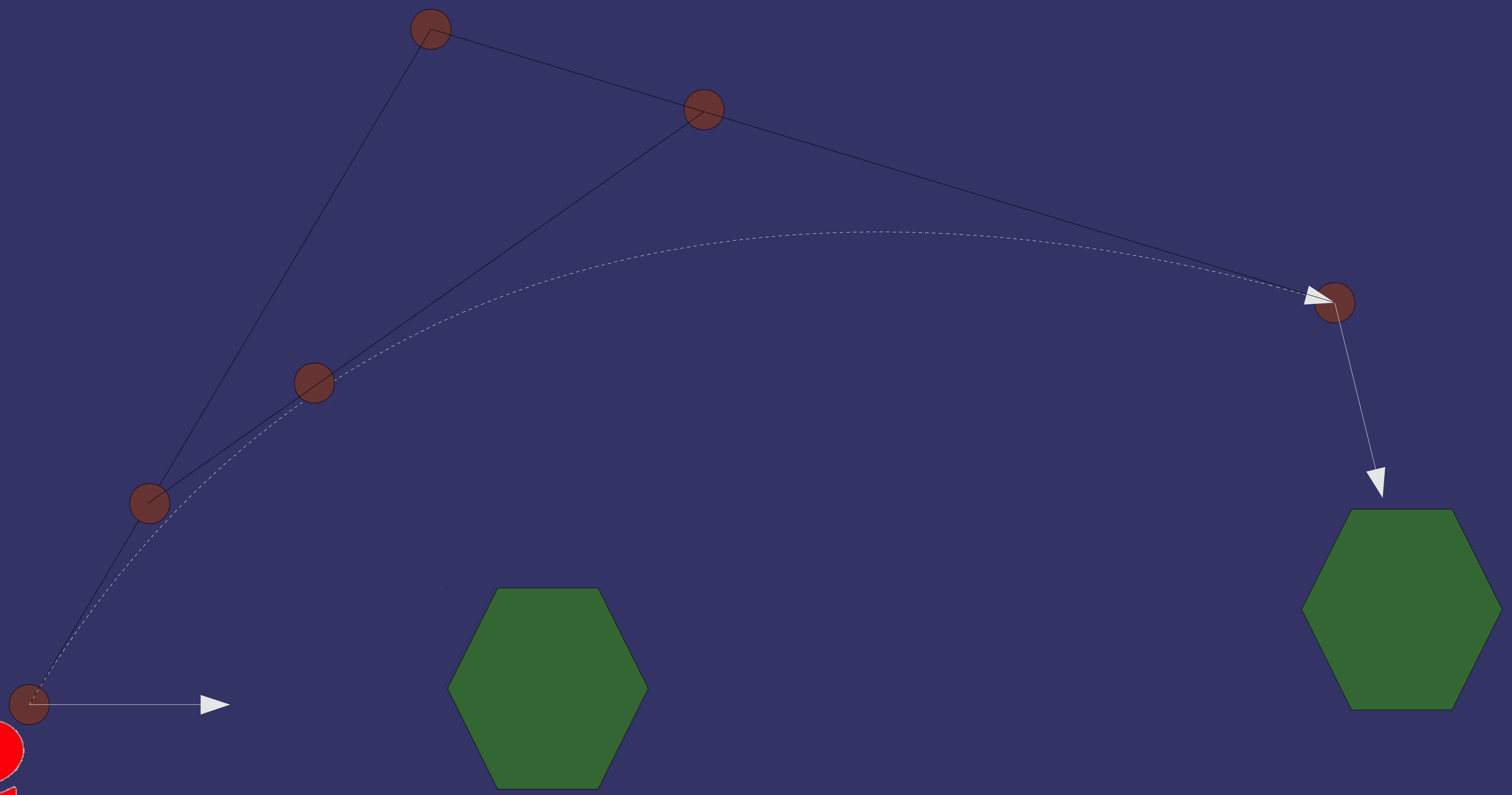
Camera Control

- p(0.3) = lerp(p_0, p_1, 0.3) lerp(p_1, p_2, 0.3)



Camera Control

- $p(0.3) = \mathrm{lerp}(\mathrm{lerp}(p_0, p_1, 0.3), \mathrm{lerp}(p_1, p_2, 0.3), 0.3)$



6-October-2010

Bézier Curve

- This works out to:

$$\mathbf{p}(t) = (1-t)^2 \mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2 \mathbf{p}_2$$

- More formally:

$$\mathbf{p}_i^k(t) = (1-t)\mathbf{p}_i^{k-1}(t) + t\,\mathbf{p}_{i+1}^{k-1}(t), \begin{cases} \mathrm{k} & = & 1..n \\ \mathrm{i} & = & 0..n-k \end{cases}$$

  - Curve with x control points is degree x-1
    - n is the degree of the polynomial that defines the curve
    - Our curve with 3 control points is degree 2
  - The initial control points are $\mathbf{p}_i^0$ but are written $\mathbf{p}_i$

- Pronounced *beh-zee-eh*

Bézier Curve

- This works out to:

$$\mathbf{p}(t) = (1-t)^2 \mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2 \mathbf{p}_2$$

- More formally:

$$\mathbf{p}_i^k(t) = (1-t)\mathbf{p}_i^{k-1}(t) + t\,\mathbf{p}_{i+1}^{k-1}(t), \begin{cases} \mathrm{k} & = & 1..n \\ \mathrm{i} & = & 0..n-k \end{cases}$$

  - Curve with x control points is degree x-1
    - n is the degree of the polynomial that defines the curve
    - Our curve with 3 control points is degree 2
  - The initial control points are $\mathbf{p}_i^0$ but are written $\mathbf{p}_i$

Bézier Curve

- Note:
  - Curve lies within the convex hull of the control points
  - Curve only passes through $\mathbf{p}_0$ and $\mathbf{p}_n$

Bézier Curve

- Repeated interpolation is cumbersome
  - Also inefficient for large n
- Can we do better?

Bézier Curve

- Repeated interpolation is cumbersome
  - Also inefficient for large n
- Can we do better?
  - Yes!
  - We can use *algebra* instead of interpolation

Bézier Basis Functions

- Rewrite a weighted sum of control points:

$$\mathbf{p}(t) = \sum_{i=0}^{n} \mathrm{B}_i^n(t)\,\mathbf{p}_i$$

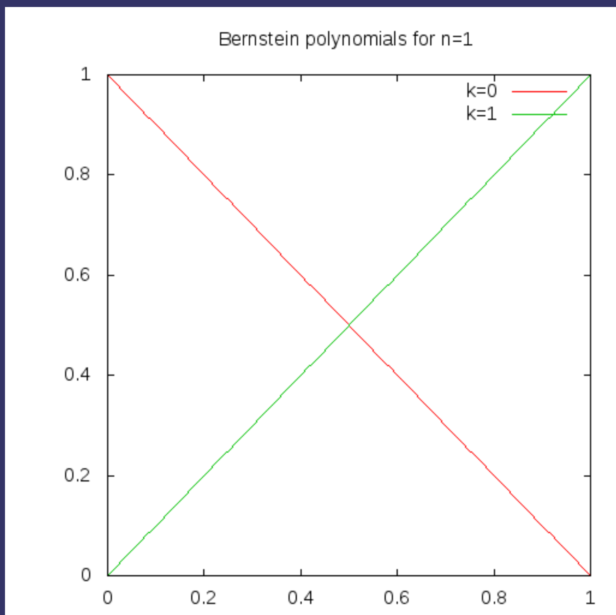
$$\begin{aligned} \mathrm{B}_i^n(t) &= \binom{n}{i} t^i (1-t)^{n-i} \\ &= \frac{n!}{i!(n-i)!} t^i (1-t)^{n-i} \end{aligned}$$

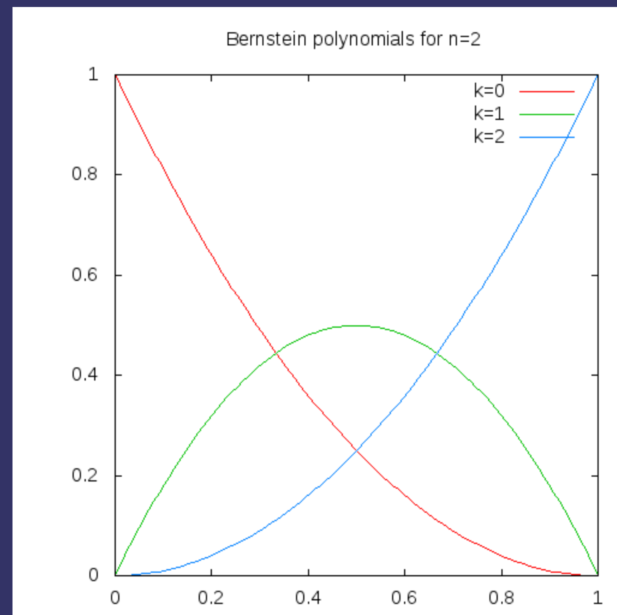
  - B_i^n is the *Bernstein polynomial* or *Bézier basis function*
  - Note:

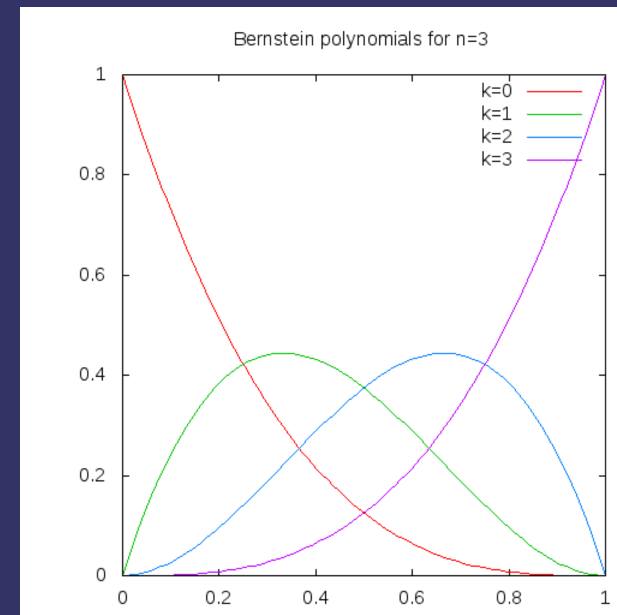
$$t \in [0,1] \rightarrow B_i^n(t) \in [0,1]$$

$$\sum_{i=0}^{n} B_i^n(t) = 1$$

Bézier Basis Functions







6-October-2010

Bézier Curve

- Usually unnecessary to go higher than n=3
  - Why?
    - What can a cubic polynomial do that a quadratic cannot?

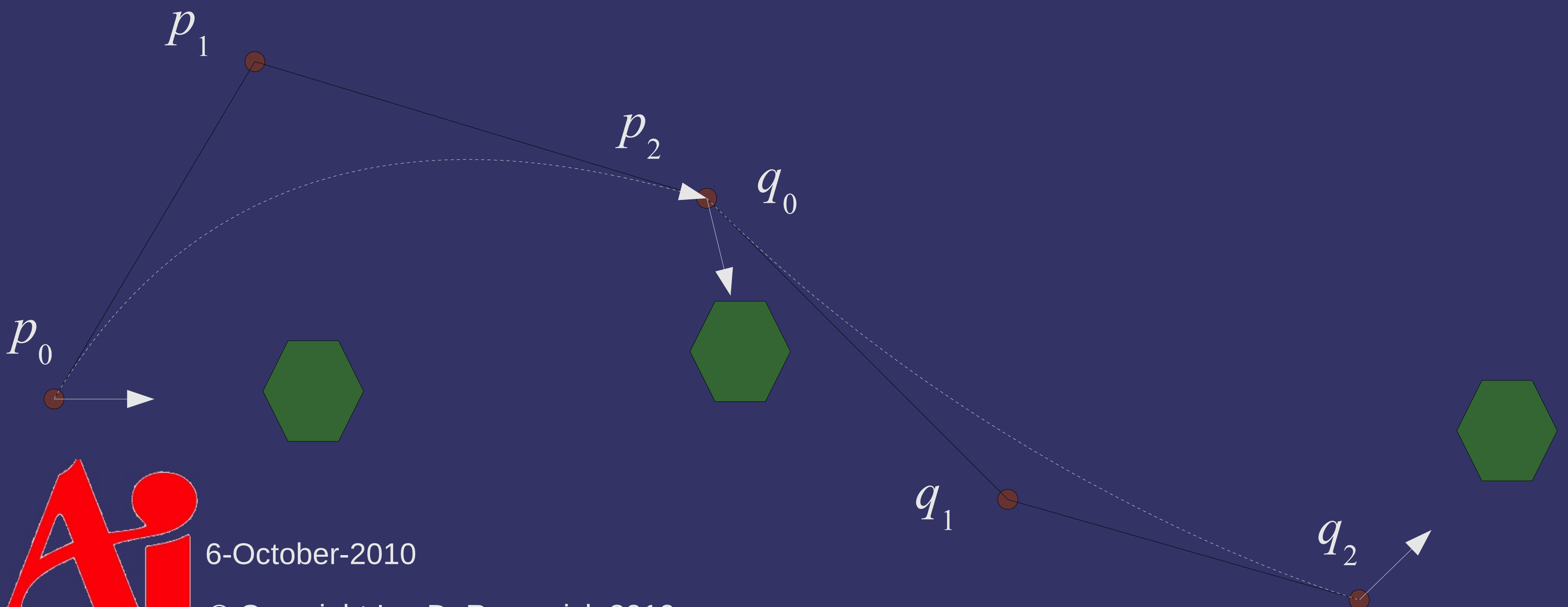
Bézier Curve

- Usually unnecessary to go higher than n=3
  - Why?
    - What can a cubic polynomial do that a quadratic cannot?
  - Cubic polynomials are the lowest degree whose derivative can change direction
    - Multiple cubic Bézier curves can be combined to approximate most shapes
  - Evaluation cost increases as n increases

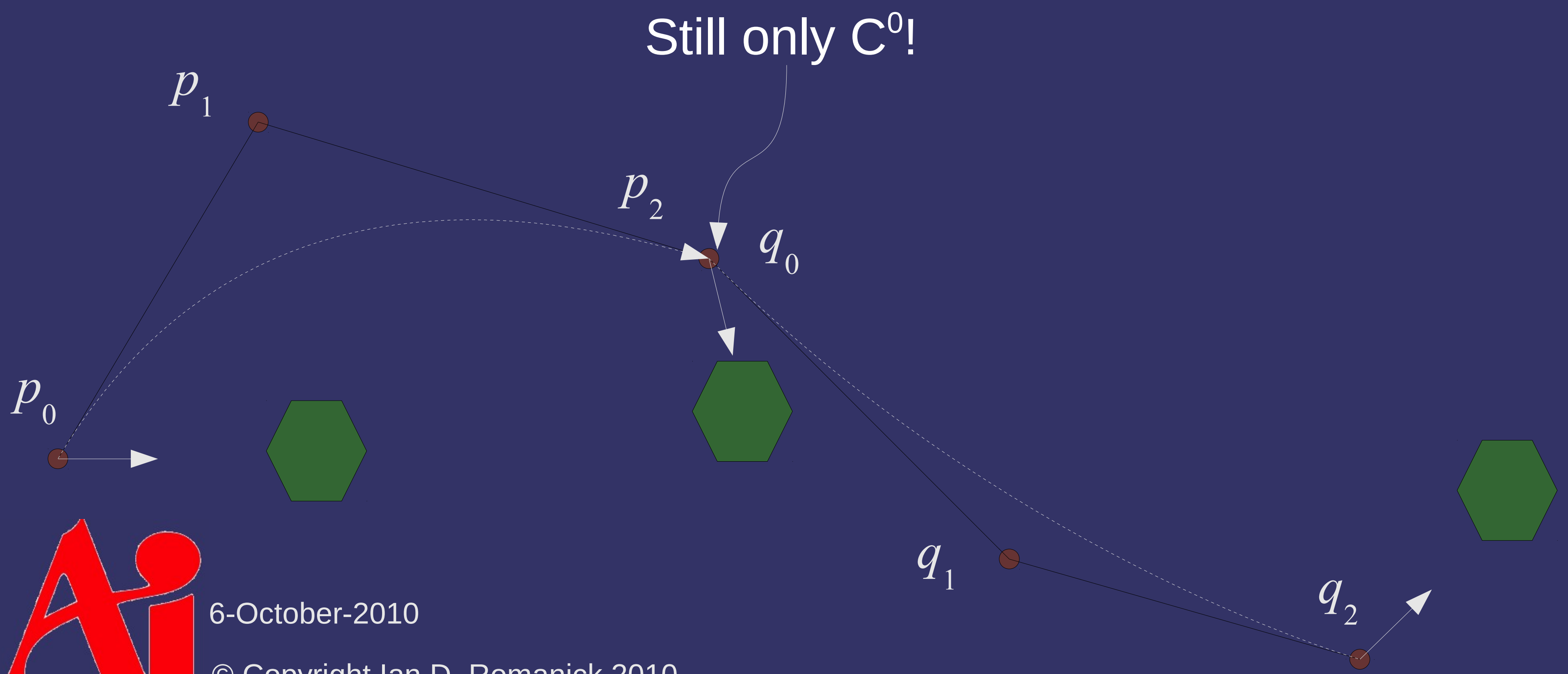
Piecewise Bézier Curves

- Curve only passes through $\mathbf{p}_0$ and $\mathbf{p}_n$
  - For camera control, we need to hit other definable points
- Define multiple curves
  - Control points $\mathbf{p}_i$, $\mathbf{q}_i$, $\mathbf{r}_i$, etc.
  - Set $\mathbf{p}_n = \mathbf{q}_0$
    - This is called a *joint*

Piecewise Bézier Curves



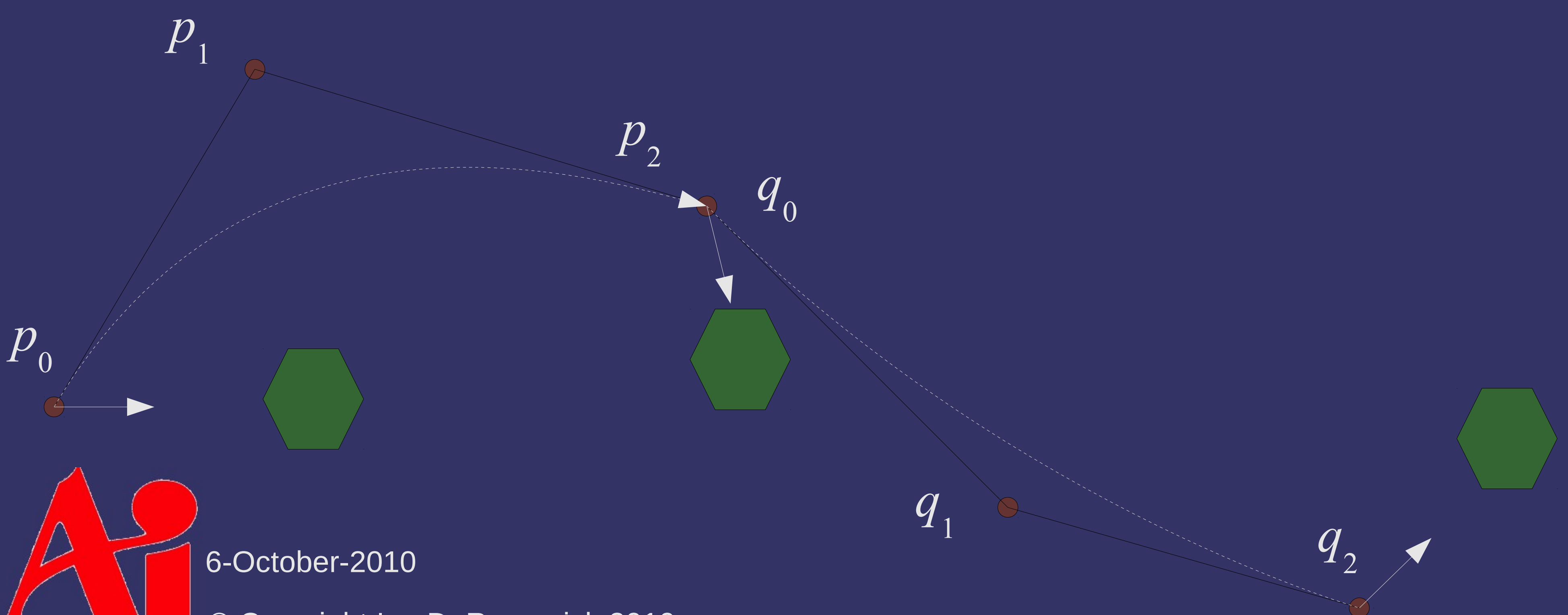
Piecewise Bézier Curves



6-October-2010

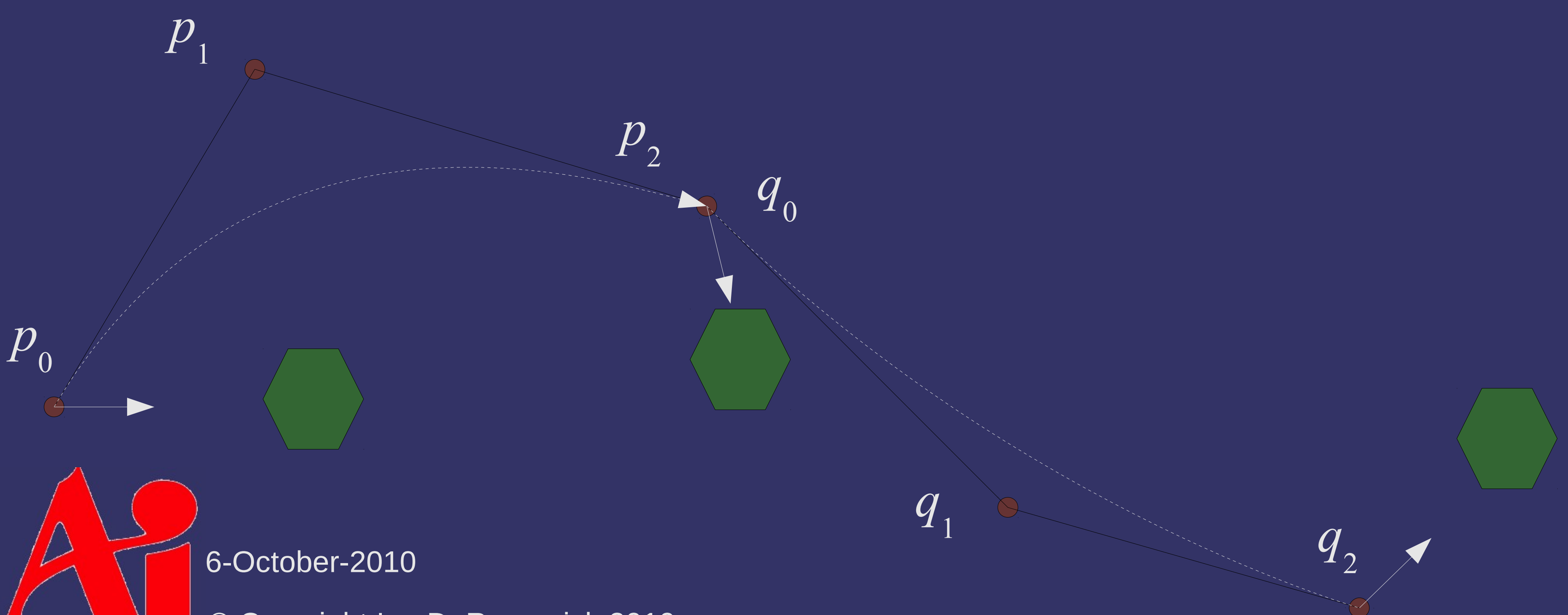
Piecewise Bézier Curves

- The piecewise function must be differentiable at the joint to be C^1
- What are the derivatives of $\mathbf{p}$ and $\mathbf{q}$ at the joint?



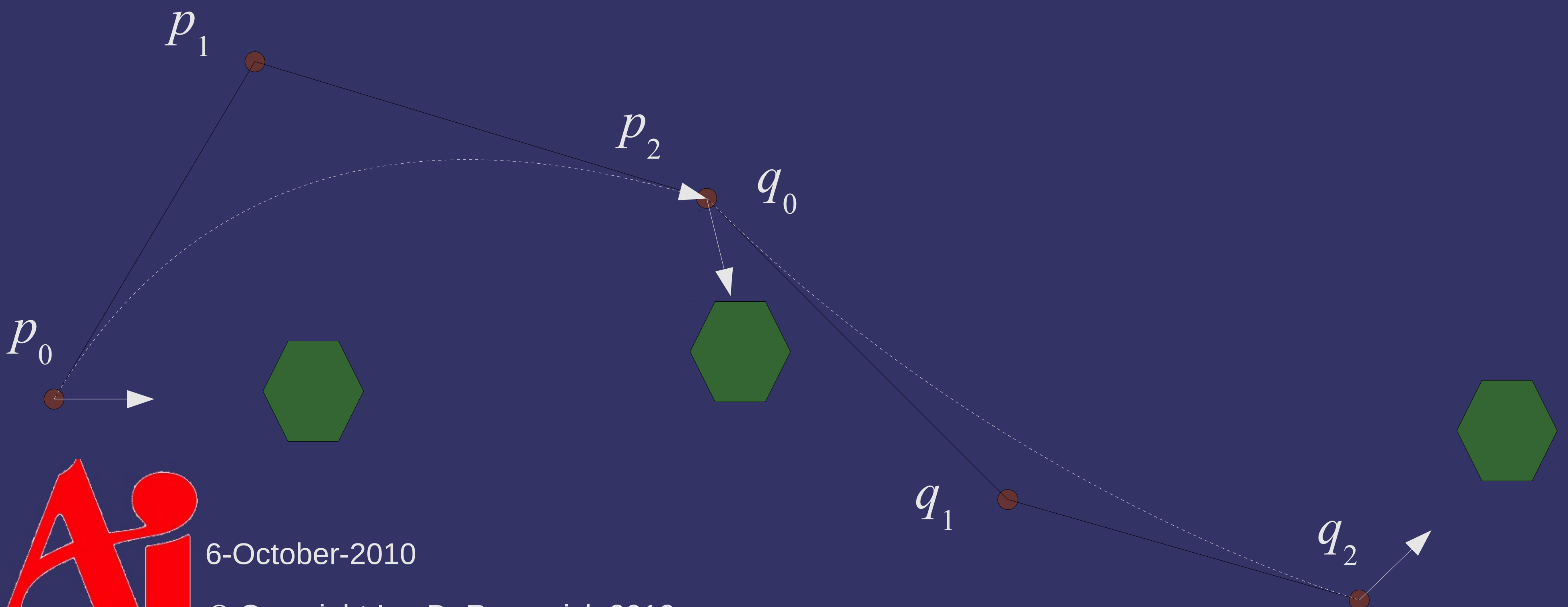
Piecewise Bézier Curves

- The piecewise function must be differentiable at the joint to be C^1
- What are the derivatives of $\mathbf{p}$ and $\mathbf{q}$ at the joint?
  - $\mathbf{p}'(1) = \mathbf{q}'(0)$



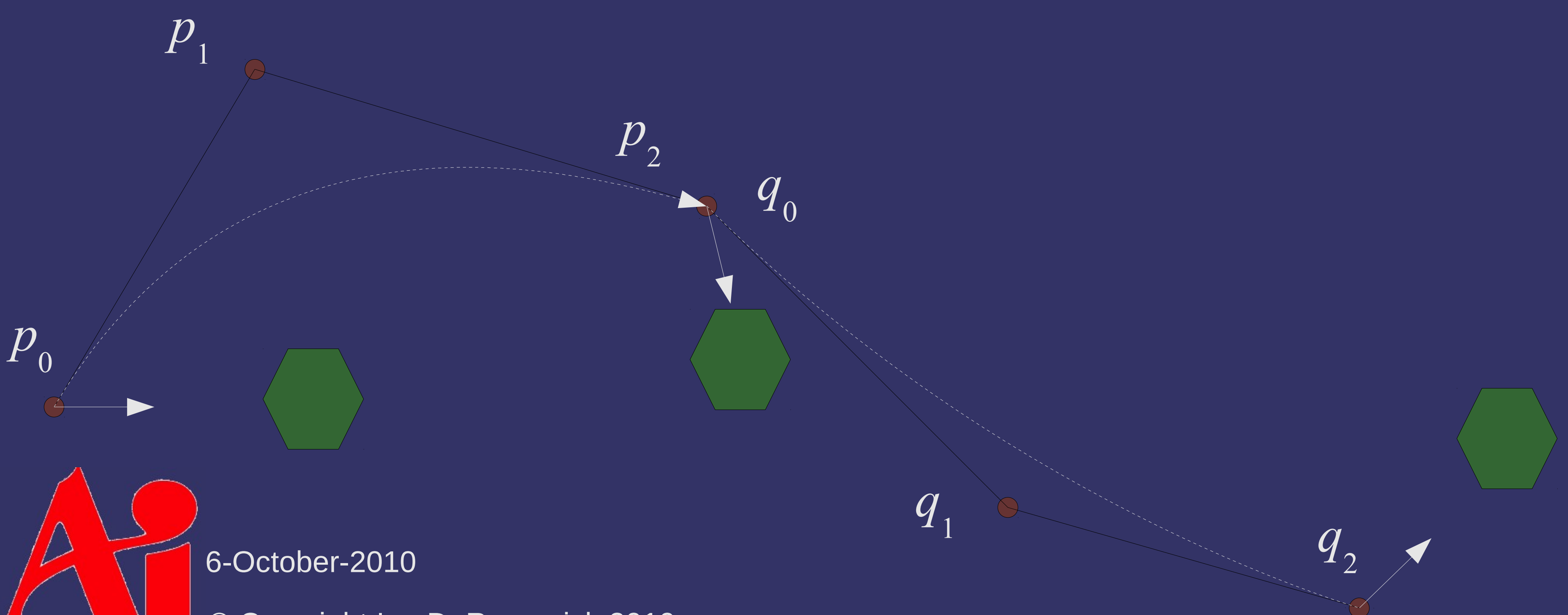
Piecewise Bézier Curves

- Generally, what is the value of the derivative of $\mathbf{f}$ at x?



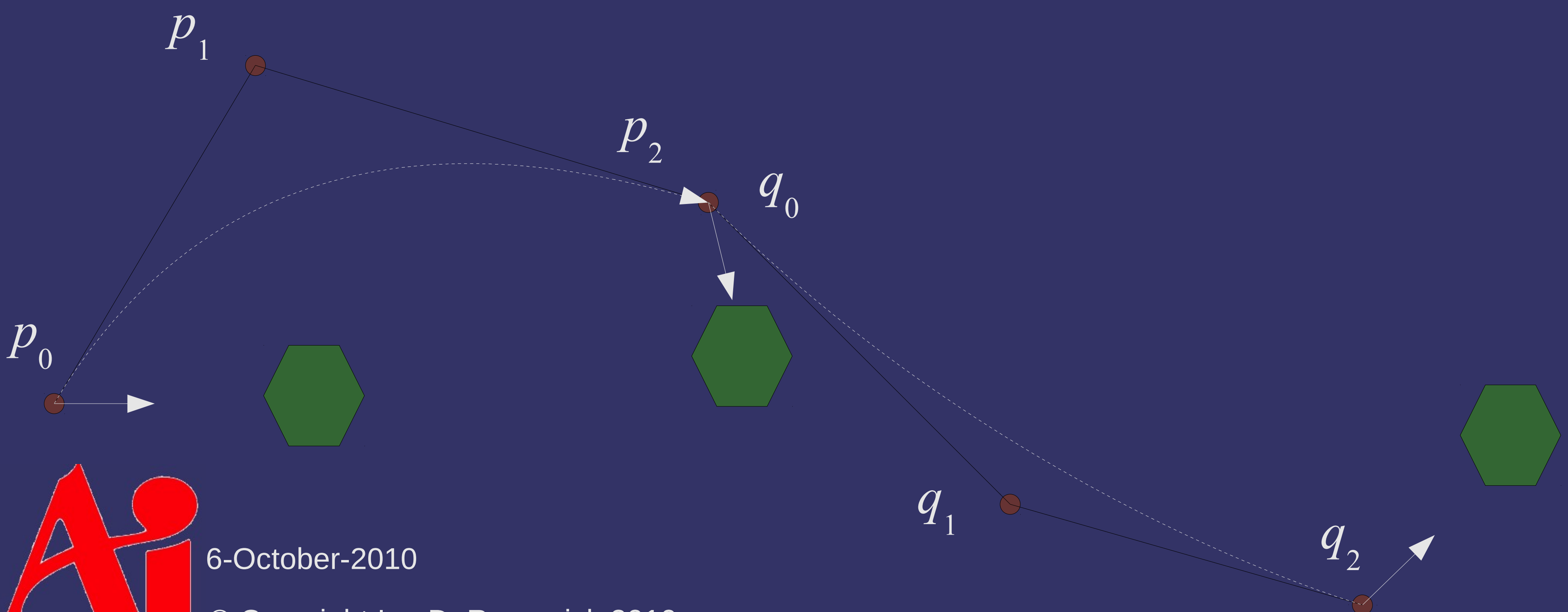
Piecewise Bézier Curves

- Generally, what is the value of the derivative of $\mathbf{f}$ at x?
  - The slope of a line *tangent* to $\mathbf{f}$ at x



Piecewise Bézier Curves

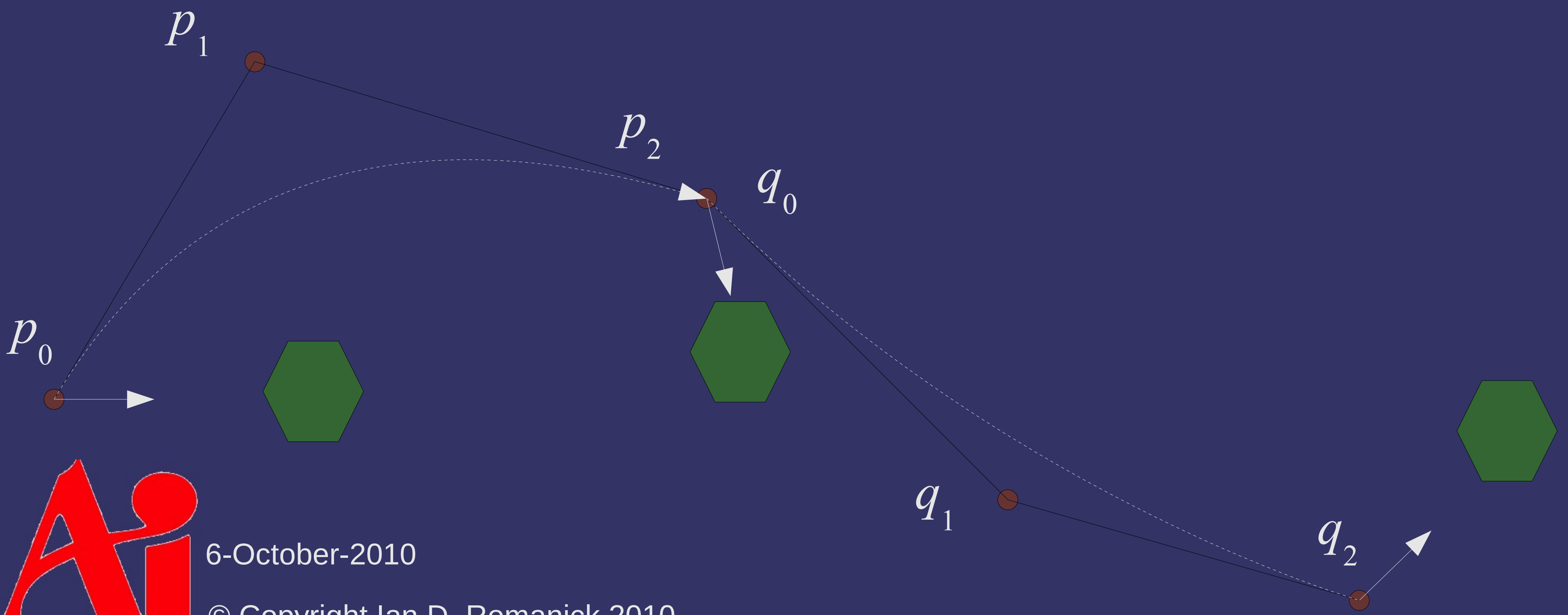
- What line is tangent to $\mathbf{p}$ at p_2?



6-October-2010

Piecewise Bézier Curves

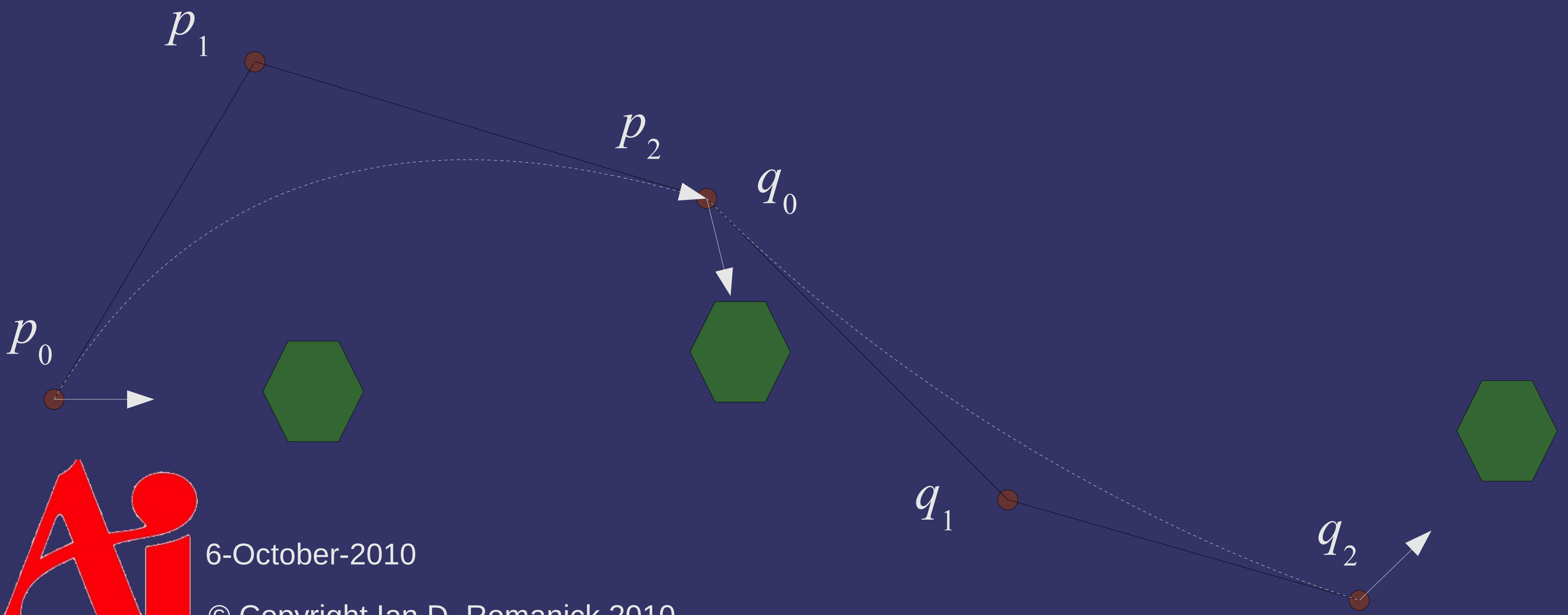
- What line is tangent to $\mathbf{p}$ at p_2?
  - $\overline{p_1 p_2}$



6-October-2010

Piecewise Bézier Curves

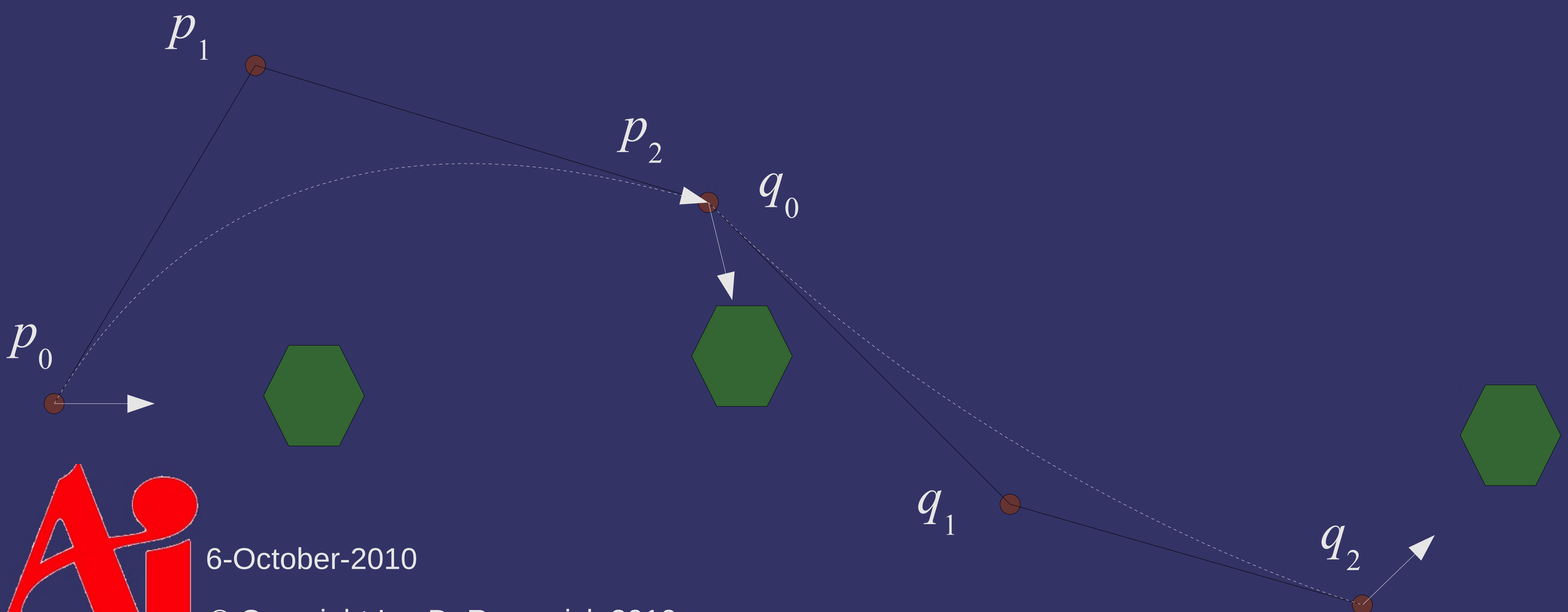
- How can we achieve C^1?



6-October-2010

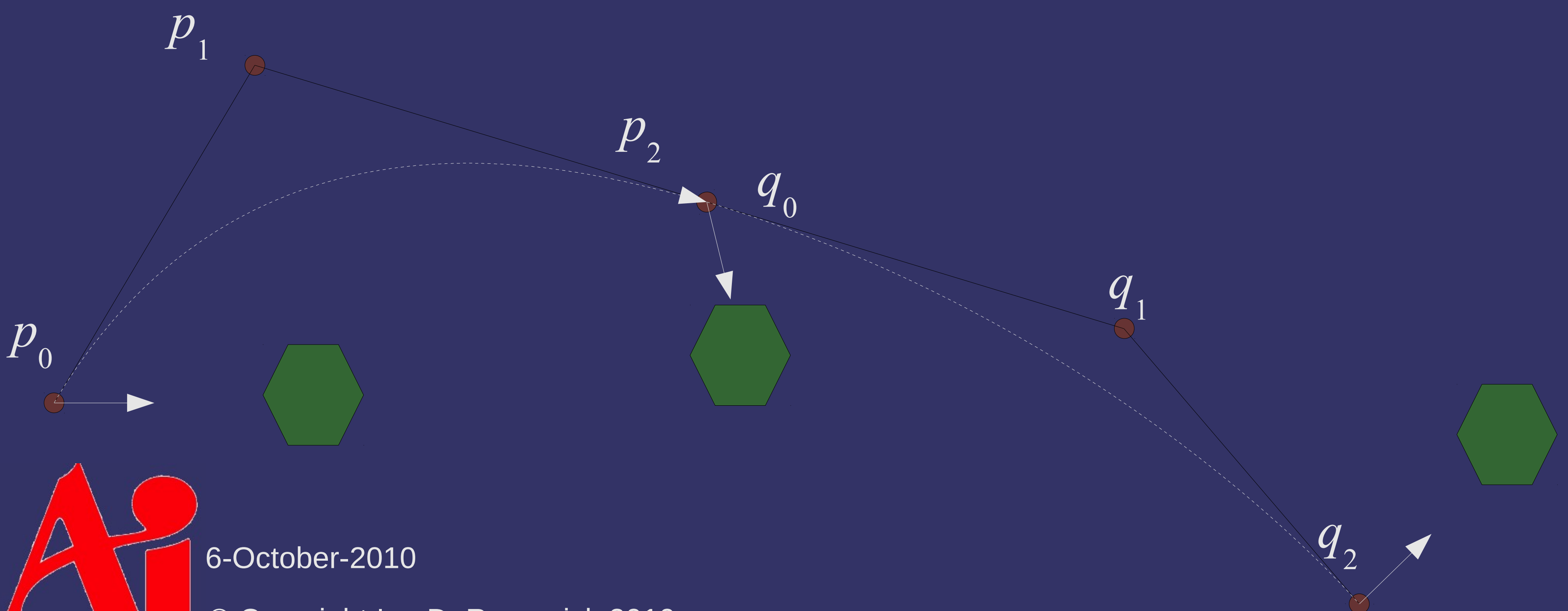
Piecewise Bézier Curves

- How can we achieve C^1?
  - Move p_1 and / or q_1 so that $\overline{p_1 p_2}$ and $\overline{q_0 q_1}$ are parallel



Piecewise Bézier Curves

- How can we achieve C[1]?
  - Move p_1 and / or q_1 so that $\overline{p_1 p_2}$ and $\overline{q_0 q_1}$ are parallel
  - This can dramatically change the curves



Piecewise Bézier Curves

- If $|\mathbf{m}_1| \neq |\mathbf{m}_2|$ there will be a speed change at the joint
  - This is *not* C^1, but it's better than C^0
  - Sometimes G^1 for *geometrical continuity*

Derivative of a Bézier Curve

- Derivative using the sum rule and regrouping:

$$\frac{d}{dt}\mathbf{p}(t) = n\sum_{i=0}^{n-1} B_i^{n-1}(t)(\mathbf{p}_{i+1} - \mathbf{p}_i)$$

  - Exercise for the reader to confirm:

$$\frac{d}{dt}\mathbf{p}(0) = \mathbf{p}_1 - \mathbf{p}_0$$

$$\frac{d}{dt}\mathbf{p}(1) = \mathbf{p}_n - \mathbf{p}_{n-1}$$

  - Result is a Bézier curve of one lower degree

Curved Surfaces

- Start with the same interpolation games
  - First extend from one parameter, t, to two parameters $\langle u, v\rangle$
  - Use four control points, $\mathbf{p}_{00}$, $\mathbf{p}_{01}$, $\mathbf{p}_{10}$, $\mathbf{p}_{11}$, instead of two
  - Interpolate between adjacent pairs:

$$\begin{aligned}
\mathbf{e} &= (1-u)\mathbf{p}_{00}+v\,\mathbf{p}_{01}\\
\mathbf{f} &= (1-u)\mathbf{p}_{10}+v\,\mathbf{p}_{11}\\
\mathbf{p}(u,v) &= (1-v)\mathbf{e}+v\,\mathbf{f}\\
&= (1-u)(1-v)\mathbf{p}_{00}+u(1-v)\mathbf{p}_{01}+(1-u)v\,\mathbf{p}_{10}+uv\,\mathbf{p}_{11}
\end{aligned}$$

  - Also known as *bilinear interpolation*

Curved Surfaces

- Extend to a curved surface in the same way as extending a line to a curve:
  - Add control points
    - For an $n \times m$ degree patch, there are $(n+1)(m+1)$ control points
    - Usually $n = m$
  - Recursively interpolate between the control points
    - Or use Bernstein form

Bézier Patches

- Bernstein form:

$$\mathbf{p}(u,v) = \sum_{i=0}^{m} \mathrm{B}_i^m(u) \sum_{j=0}^{n} \mathrm{B}_j^n(v)\, \mathbf{p}_{i,j}$$

  - As with Bézier curves:
    - Surface lies within convex hull of control points
    - And:

$$(u,v) \in [0,1] \times [0,1] \rightarrow B_i^m(u) B_j^n(v) \in [0,1]$$

$$\sum_{i=0}^{m} \sum_{j=0}^{n} B_i^m(u) B_j^n(v) = 1$$

    - Second summation is just a Bézier curve!

Bézier Patches

- Bernstein form:

$$\mathbf{p}(u,v)=\sum_{i=0}^{m}\mathrm{B}_i^m(u)\boxed{\sum_{j=0}^{n}\mathrm{B}_j^n(v)\,\mathbf{p}_{i,j}}$$

  – As with Bézier curves:
    – Surface lies within convex hull of control points
    – And:

$$(u,v)\in[0,1]\times[0,1]\rightarrow B_i^m(u)\,B_j^n(v)\in[0,1]$$

$$\sum_{i=0}^{m}\sum_{j=0}^{n}B_i^m(u)\,B_j^n(v)=1$$

    – Second summation is just a $\boxed{\text{Bézier curve!}}$

Derivative of a Bézier Patch

- Similar to Bézier curves:

$$\frac{\partial \mathbf{p}(u,v)}{\partial u} = m \sum_{j=0}^{n} \sum_{i=0}^{m-1} B_i^{m-1}(u) B_j^n(v) [\mathbf{p}_{i+1,j} - \mathbf{p}_{i,j}]$$

$$\frac{\partial \mathbf{p}(u,v)}{\partial v} = n \sum_{i=0}^{m} \sum_{j=0}^{n-1} B_i^m(u) B_j^{n-1}(v) [\mathbf{p}_{i,j+1} - \mathbf{p}_{i,j}]$$

Normals of a Bézier Patch

- How do we calculate the normal?
  - What we *really* want is the normal of the plane tangent to the surface

Normals of a Bézier Patch

- How do we calculate the normal?
  - What we *really* want is the normal of the plane tangent to the surface
  - The partial derivatives give two vectors that lie in that plane... just take the cross product!

$$\mathbf{n}(u,v) = \frac{\partial \mathbf{p}(u,v)}{\partial u} \times \frac{\partial \mathbf{p}(u,v)}{\partial v}$$

Phong Shading Recap

- Phong shading... aka per-fragment lighting
  - Calculate lighting parameters per-vertex
  - Interpolate calculated values
  - Calculate lighting per-fragment based on interpolated parameter values

Phong Shading Recap

```
attribute vec3 normal;
uniform mat3 normal_xform;
uniform mat4 vertex_xform;
uniform mat4 mvp;

varying vec3 normal_es;
varying vec3 pos_es;

void main(void)
{
    gl_Position = mvp * gl_Vertex;

    normal_es = normal_xform * normal;
    pos_es = vertex_xform * gl_Vertex;
}
```

Phong Shading Recap

```
uniform vec3 light_pos_es;
uniform vec4 diff_color;
varying vec3 normal_es;
varying vec3 pos_es;
const vec3 eye_es = vec3(0);

void main(void)
{
    vec3 l = normalize(light_pos_es – pos_es);
    vec3 v = normalize(eye_es - pos_es);
    vec3 h = normalize(l + v);
    float n_dot_l = dot(normal_es, l);
    vec4 diff = diff_color * n_dot_l;
    float spec = pow(dot(n, h), 16.0);

    gl_FragColor = step(0.0, n_dot_l) *
        vec4(diff.xyz + vec3(spec), 1.0);
}
```

Surface-Space

- From the point of view of the surface, what is the normal vector?
  - We'll call this *surface-space*

Surface-Space

- From the point of view of the surface, what is the normal vector?
  - We'll call this *surface-space*
  - Assuming the surface is flat, $\mathbf{n}_{surf} = (0, 0, 1)$

Surface-Space

- If we know $\mathbf{n}_{world}$, can we create transformation that will generate $\mathbf{n}_{surf}$?
  - Not uniquely
    - An orthonormal basis requires three orthogonal, normalized vectors, but we only have one
      - If we have two we can generate the third
    - This is the same reason we need the “up” vector to create the camera look-at transform
  - If only we had another vector in plane...

Surface-Space

- Create a new vector, and call it the *tangent*
  - Either partial derivative of a Bézier patch can be used for $\mathbf{t}_{surf}$
    - Usually $\partial\mathbf{p}/\partial u$ is used
  - Knowing $\mathbf{n}_{surf}$ and $\mathbf{t}_{surf}$ is enough to create an orthonormal basis
  - This basis can transform *any* vector to surface-space from object-space
    - $\mathbf{n}_{obj}$ is an obvious choice
    - For lighting, $\mathbf{v}$ and $\mathbf{l}$ need to be in the same space as $\mathbf{n}$
- Because the tangent vector is used, surface-space is sometimes called *tangent-space*

Surface-Space

```
varying vec3 light_ss;
varying vec3 eye_ss;
attribute vec3 tangent;
attribute vec3 normal;

void main(void)
{
    gl_Position = mvp * gl_Vertex;

    vec3 tangent_es = normal_xform * tangent;
    vec3 normal_es = normal_xform * normal;
    vec3 bitangent_es = cross(normal_es, tangent_es);
    mat3 tbn = mat3(tangent_es, bitangent_es, normal_es);

    vec3 pos_es = vec3(vertex_xform * gl_Vertex);
    vec3 light_es = light_pos_es - pos_es;

    light_ss = normalize(light_es * tbn);
    eye_ss = -normalize(pos_es * tbn);
}
```

Surface-Space

```
varying vec3 light_ss;
varying vec3 eye_ss;
attribute vec3 tangent;
attribute vec3 normal;

void main(void)
{
    gl_Position = mvp * gl_Vertex;

    vec3 tangent_es = normal_xform * tangent;
    vec3 normal_es = normal_xform * normal;
    vec3 bitangent_es = cross(normal_es, tangent_es);
    mat3 tbn = mat3(tangent_es, bitangent_es, normal_es);

    vec3 pos_es = vec3(vertex_xform * gl_Vertex);
    vec3 light_es = light_pos_es - pos_es;

    light_ss = normalize(light_es * tbn);
    eye_ss = -normalize(pos_ss * tbn);
}
```

This actually calculates $\mathbf{M}_s^T$

Surface-Space

```
varying vec3 light_ss;
varying vec3 eye_ss;
attribute vec3 tangent;
attribute vec3 normal;

void main(void)
{
    gl_Position = mvp * gl_Vertex;

    vec3 tangent_es = normal_xform * tangent;
    vec3 normal_es = normal_xform * normal;
    vec3 bitangent_es = cross(normal_es, tangent_es);
    mat3 tbn = mat3(tangent_es, bitangent_es, normal_es);

    vec3 pos_es = vec3(vertex_xform * gl_Vertex);
    vec3 light_es = light_pos_es - pos_es;

    light_ss = normalize(light_es * tbn);
    eye_ss = -normalize(pos_ss * tbn);
}
```

This actually calculates $\mathbf{M}_s^T$

Remember: $\mathbf{Mv} = \mathbf{vM}^T$

Surface-Space

```
varying vec3 light_ss;
varying vec3 eye_ss;
uniform vec4 diff_color;

void main(void)
{
    vec3 l = normalize(light_ss);
    vec3 v = normalize(eye_ss);
    vec3 h = normalize(l + v);
    float n_dot_l = l.z;
    vec4 diff = diff_color * n_dot_l;
    float spec = pow(h.z, 16.0);

    gl_FragColor = step(0.0, n_dot_l) *
        vec4(diff.xyz + vec3(spec), 1.0);
}
```

Surface-Space

```
varying vec3 light_ss;
varying vec3 eye_ss;
uniform vec4 diff_color;

void main(void)
{
    vec3 l = normalize(light_ss);
    vec3 v = normalize(eye_ss);
    vec3 h = normalize(l + v);
    float n_dot_l = l.z;
    vec4 diff = diff_color * n_dot_l;
    float spec = pow(h.z, 16.0);

    gl_FragColor = step(0.0, n_dot_l) *
        vec4(diff.xyz + vec3(spec), 1.0);
}
```

Remember: **n** is (0, 0, 1)!

Surface-Space

- What is $\mathbf{b}$?
  - In the calculation: $\mathbf{b} = \mathbf{n} \times \mathbf{t}$
  - Correctly, this is the *bi-tangent*
    - Many places incorrectly call it the bi-normal
    - Either way, we'll just call it $\mathbf{b}$
  - Generally easier and more efficient to compute this in a shader than supply it as an input
    - We *cannot* just use $\partial\mathbf{p}/\partial v$ from from our surface evaluation because the two partial derivatives may not be orthogonal to each other!

Surface-Space

- What does this math headache gain us?
  - Just a trivial fragment shader optimization so far
    - Seems hardly worth it
  - What else?

Bump Mapping

- What if the surface isn't really flat or smoothly curved?
  - Just like few real surfaces have truly uniform color, few real surfaces have uniform normals
  - Use the same solution!
    - Store colors in an image → store normals in an image

Normal Map Storage

- Store the X, Y, and Z values of the surface-space normals in the R, G, and B components
  - Since Z tends to be close to 1.0, these images tend to look very blue

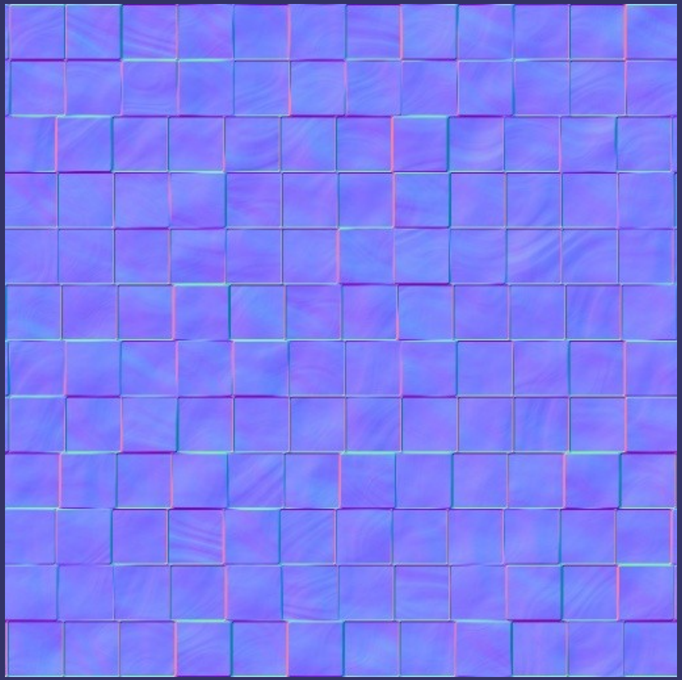


Image from http://www.filterforge.com/filters/243-normal.html

Normal Map Storage

- What is the range of colors in a texture?

Normal Map Storage

- What is the range of colors in a texture?
  - [0.0, 1.0]
  - We have to convert these to the [-1, 1] range desired for normal directions
    - Just convert X and Y... Z must be > 0, so just leave it

Normal Map Storage

- We don't even need Z
  - Z must always be > 0.0
  - Derive it from X and Y:

Normal Map Storage

- We don't even need Z
  - Z must always be > 0.0
  - Derive it from X and Y:

$$\sqrt{x^2+y^2+z^2}=1.0$$
$$x^2+y^2+z^2=1.0$$
$$z^2=1.0-x^2-y^2$$
$$z=\sqrt{1.0-x^2-y^2}$$

Normal Map Storage

- 2-component textures can be achieved in a couple ways:
  - Use `GL_LUMINANCE_ALPHA`
    - Some hardware doesn't really support this, so it will silently convert it to RGBA...making it bigger
  - Use `GL_RG`
    - Requires `GL_ARB_texture_rg` or OpenGL 3.0
  - Use `GL_COMPRESSED_RED_GREEN_RGTC2_EXT`
    - Requires `GL_ARB_texture_compression_rgtc`, `GL_EXT_texture_compression_rgtc`, or OpenGL 3.0
    - May add undesired compression artifacts

References

Lengyel, Eric. “Computing Tangent Space Basis Vectors for an Arbitrary Mesh”. Terathon Software 3D Graphics Library, 2001. http://www.terathon.com/code/tangent.html

Normal map photography tutorial:

http://www.zarria.net/nrmphoto/nrmphoto.html

OpenGL extension specs:

http://www.opengl.org/registry/specs/ARB/texture_rg.txt

http://www.opengl.org/registry/specs/ARB/texture_compression_rgtc.txt

Next week...

- Render-to-texture
- Environment mapping
  - Rendering to env maps
- Improving the reflection model
  - Using env maps as better lights
  - Fresnel reflection
- Read:

Michael Toksvig. “Mipmapping Normal Maps.” http://developer.nvidia.com/object/mipmapping_normal_maps.html

Real-Time Rendering 3rd Edition, chapter 13.1 and 13.2.

Legal Statement

This work represents the view of the authors and does not necessarily represent the view of Intel or the Art Institute of Portland.

OpenGL is a trademark of Silicon Graphics, Inc. in the United States, other countries, or both.

Khronos and OpenGL ES are trademarks of the Khronos Group.

Other company, product, and service names may be trademarks or service marks of others.