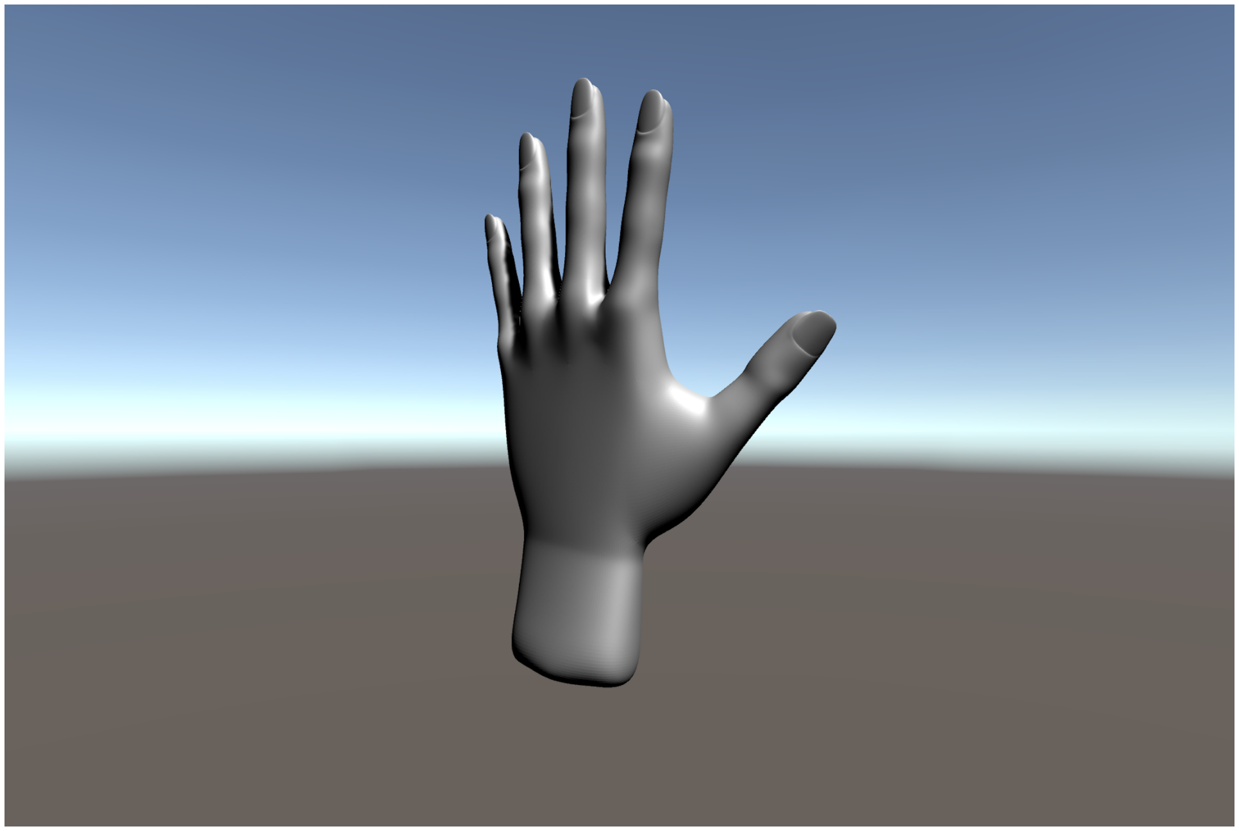
Exercise 3 - Subdivision Surfaces & Bézier Curves

In this exercise you will implement the Catmull-Clark subdivision algorithm and create an interactive chain model using Bézier curves.

The goal of this exercise is to learn about Bézier curves, subdivision surfaces and working with mesh data structures.

You **must** submit this exercise in pairs.



EX3 Guidelines

- In this exercise you may only edit the files CatmullClark.cs, BezierCurve.cs, BezierMesh.cs and Chain.cs, according to the instructions.
- You are given a folder of different OBJ mesh files which you can use to check your subdivision code.
- You may **not** change or add properties to the given classes `CCMeshData` and `QuadMeshData`.

General Guidelines

You may lose points for not following these guidelines.

- Make sure you are using **Unity 2020.1.6f1**
- Make sure that you understand the effect of each part of your code
- Make sure that your code does what it's supposed to do and that your results look the way they should
- Keep your code readable and clean! Avoid code duplication, comment non-trivial code and preserve coding conventions
- Keep your code efficient

Submission

Submit a single `.zip` file containing **only** the following files:

- **CatmullClark.cs**
- **BezierCurve.cs**
- **BezierMesh.cs**
- **Chain.cs**
- **`readme.txt`** that includes both partners' IDs and usernames. List the URLs of web pages that you used to complete this exercise, as well as the usernames of all students with whom you discussed this exercise

Deadline

Submit your solution via the course's moodle no later than **Sunday, December 12 at 23:55**.

Late submission will result in 2^{N+1} points deduction where N is the number of days between the deadline and your submission. The minimum grade is 0, saturday is excluded.

Part 1 - Catmull-Clark subdivision

Part 1.0 / Setup

1. Download the exercise zip file from the course Moodle website and unzip it somewhere on your computer.
2. In Unity Hub, go to *Projects* and click the *Add* button on the top right. Select the folder that you have downloaded.
3. Open the project. Once Unity is open, double click the SubdivisionScene to open it.
4. Open the file CatmullClark.cs to edit it.

Note that when clicking “Subdivide” in the MeshSubdivider UI in the inspector, the function `Subdivide` of the class `CatmullClark` is called.

`QuadMeshData` documentation

In this exercise you will be working with quad-faced meshes, meaning each face contains 4 vertices rather than 3. To help you with this, you are given the class `QuadMeshData`.

`List<Vector3>` **`vertices`**

A list of mesh vertex positions

`List<Vector4>` **`quads`**

A list of quad mesh faces, where each face is defined by 4 indices of vertices in the `vertices` list

`QuadMeshData``(List<Vector3> vertices, List<Vector4> quads)`

`QuadMeshData` constructor - Initializes a `QuadMeshData` object with the given `vertices` and `quads`

`CCMeshData` documentation

In addition, you are provided with the helper class `CCMeshData` to hold the data needed for the Catmull- Clark subdivision algorithm. You will fill its properties as you implement the algorithm:

`List<Vector3>` **`points`**

Original mesh points (i.e. vertex positions)

`List<Vector4>` **`faces`**

A list of quad faces of the original mesh. Each face defined by 4 indices of vertices in the `points` list

`List<Vector4>` **`edges`**

A list of all unique edges in the mesh defined by points and faces, as explained in part 1 of this exercise

`List<Vector3>` **`facePoints`**

Face points, as described in the Catmull-Clark subdivision algorithm

`List<Vector3>` **`edgePoints`**

Edge points, as described in the Catmull-Clark subdivision algorithm

`List<Vector3>` **`newPoints`**

New locations of the original mesh points, as described in the Catmull-Clark subdivision algorithm

Part 1.1 / Getting Edges

In this part you will implement the function `GetEdges`. It receives a `CCMeshData` object, with its `points` and `faces` properties filled:

`List<Vector3>` **`points`** - a list of mesh vertex positions

`List<Vector4>` **`faces`** - a list of quad mesh faces, each face defined by 4 indices of vertices in the `points` list

- The function returns a `List<Vector4>` representing a list of all unique edges in the mesh defined by `points` and `faces`. Each edge is represented by a `Vector4` in the following format:

```
Vector4(p1, p2, f1, f2)
```

`p1`, `p2` are the 2 edge vertices, given by indices in the `point` list.

`f1`, `f2` are the faces incident to the edge, given by indices in the `faces` list. If the edge belongs to one face only, `f2` is set to -1.

Part 1.2 / Catmull-Clark Subdivision

- In this part you will complete the implementation of the function `Subdivide`. The function receives 1 parameter:

  `QuadMeshData` **`meshData`** - a vertex and face representation of a 3D quad mesh

1. Implement the Catmull-Clark subdivision algorithm as learned in class. The general steps of the algorithm must be implemented in the given functions:
   - `List<Vector3> GetFacePoints(CCMeshData mesh)`
   - `List<Vector3> GetEdgePoints(CCMeshData mesh)`
   - `List<Vector3> GetNewPoints(CCMeshData mesh)`
2. After implementing these functions, use the completed `CCMeshData` object to construct and return a `QuadMeshData` representing the input mesh after one iteration of Catmull-Clark subdivision.

- You may add any function you wish to CatmullClark.cs.
- Efficiency is important - you don't have to implement the optimal solution, but you may lose points for inefficient code.
- Note that even if your code is very efficient, after a few iterations the mesh will become so big that subdividing might freeze Unity!
- Document your code and test it with different OBJ files to make sure everything works properly.

Part 2 - Bézier Curves

Part 2.0 / Setup

1. In the project view, double click the BezierScene to open it.

2. Open the file BezierCurve.cs to edit it.

   Click the BezierMesh game object. In the scene view, take a look at the 4 control points as well as the Bézier curve they define. The bezier inspector is drawn using Unity's built-in methods. You can use the standard move tool to manipulate these points or edit them directly using the inspector UI.

Part 2.1 / Bézier Curve Implementation

In this part you will implement several functions in the class `BezierCurve`. All functions receive a parameter `float` **`t`** in the range [0,1]

1. `Vector3 GetPoint(float t)`

   Returns the position of the point $\mathrm{B}(t)$ on the bezier curve B defined by `p0`, `p1`, `p2`, `p3`.

2. `Vector3 GetFirstDerivative(float t)`

   Returns the first derivative $\mathrm{B}'(t)$ of the bezier curve defined by `p0`, `p1`, `p2`, `p3`.

3. `Vector3 GetSecondDerivative(float t)`

   Returns the second derivative $\mathrm{B}''(t)$ of the bezier curve defined by `p0`, `p1`, `p2`, `p3`.

4. `Vector3 GetTangent(float t)`

   Returns the tangent vector to the curve at point $\mathrm{B}(t)$.

5. `Vector3 GetNormal(float t)`

   Returns the normal vector to the curve at point $\mathrm{B}(t)$. Use the Frenet-Serret method as seen in class.

6. `Vector3 GetBinormal(float t)`

   Returns the binormal (also known as the bitangent) vector to the curve at point $\mathrm{B}(t)$.

Hint: test your code as you write it. You can use `Debug.DrawLine` to see the Bézier curve and associated vectors. Make sure to set a duration to see the actual line!

Part 2.2 / Bézier Mesh

1. Open the file BezierMesh.cs. In this part you will implement the function `BuildMesh`, which builds a "tube" mesh around a given Bézier curve.

   This function will be called every time the Bézier curve control points are moved in the scene view, so you can edit the mesh interactively. You can also click "Update Mesh" in the inspector.

2. Receives 3 parameters:

   `BezierCurve` **`curve`** - the Bézier curve around which the mesh will be constructed

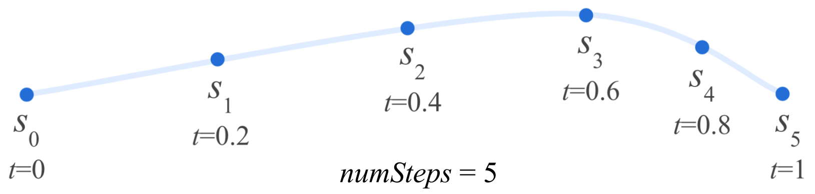
   `float` **`radius`** - the distance of mesh vertices from the actual Bézier curve

   `int` **`numSteps`** - number of sample points along the curve

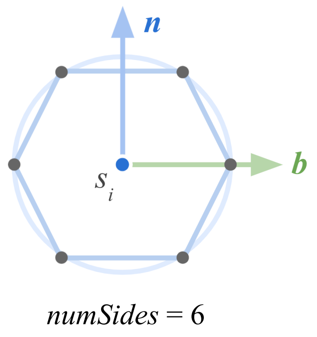
   `int` **`numSides`** - number of vertices at each sample point

3. The function returns a `Mesh` constructed in the following manner:

   i. Sample `numSteps+1` points along the given Bézier curve B. Each sample point s_i is given by $s_i = \mathrm{B}(t_i)$ where $t_i = i \,/\, numSteps$ for $i = 0, \ldots, numSteps$.



   ii. For each s_i create `numSides` vertices around the point, on the plane spanned by the normal $\boldsymbol{n}$ and binormal $\boldsymbol{b}$. The vertices should be evenly spaced on a circle centered on s_i with the given `radius`.

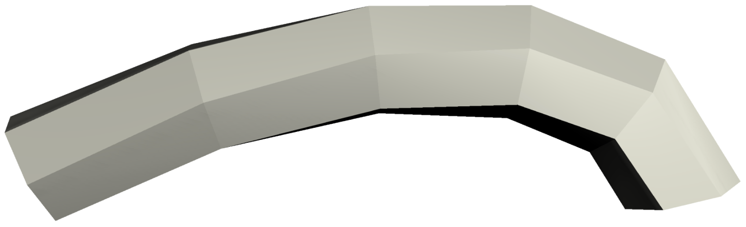


   To sample points on a circle, use the given function:

   `Vector3` **`GetUnitCirclePoint`**`(float degrees)`

   Returns a point on the unit circle, at a given angle `degrees` from the x-axis.

iii. Finally, for each segment construct `numSides` quads connecting the current sample point's vertices with the next.


Bézier curve mesh with `numSteps=5` and `numSides=6`.

Part 2.2 / Bézier Arc-Length

In this part you will calculate approximate arc-lengths of a Bézier curve using a lookup table, as seen in class. The actual LUT will be implemented as an array and is given as a class property, `float[] cumLengths`.

Open the file BezierCurve.cs and implement the following functions:

1. `void CalcCumLengths(float t)`

   Fills in the cumulative lengths lookup table `cumLengths` in the following manner:

   i. Sample `numSteps+1` points along the given Bézier curve B. Each sample point s_i is given by $s_i = \mathrm{B}(t_i)$ where $t_i = i\,/\,numSteps$ for $i = 0, \ldots, numSteps$.

   ii. Fill in the cumulative lengths array in the following manner:

   For $i = 0$: $cumLength_0 = 0$

   For $i > 0$: $cumLength_i = \sum_{j=1}^{i} ||s_j - s_{j-1}||$

2. `float ArcLength()`

   Returns the total (approximate) arc-length of the Bézier curve.

3. `float ArcLengthToT(float a)`

   Returns a `float` $t \in [0,1]$ such that $cumLength(\mathrm{B}(t)) = a$. The value of t is calculated approximately, using the `cumLength` lookup table:

   i. Find the index i such that $cumLength_i \leq a \leq cumLength_{i+1}$

ii. Return a linear interpolation between t_i and t_{i+1} to approximate the t value we are looking for: $\frac{t - t_i}{a - cumLength_i} = \frac{t_{i+1} - t_i}{cumLength_{i+1} - cumLength_i}$

You may use the functions `Mathf.Lerp` and `Mathf.InverseLerp` to perform linear interpolation between the values.

Part 2.3 / Chain Modelling

1. In the project view, double click the ChainScene to open it. take a look at the "ChainLink" game object which is constructed of 2 Bézier meshes that you have implemented in part 2.2.

2. Open the file Chain.cs and implement the method:

   `void ShowChain()` - constructs a chain made of links along the given Bezier curve.

   - Each link should be equally spaced from its neighbors on the chain, using a constant arc-length `LinkSize`.
   - The created GameObjects should be added to the `List<GameObject> chainLinks` class property.
   - Each link's local (object-space) forward direction should be aligned with the Bézier curve's tangent direction. The local up direction should alternate between the normal and binormal directions, allowing the links to fit together.
   - You may use the given function:

     `GameObject CreateChainLink(Vector3 position, Vector3 forward, Vector3 up)`

     Instantiates & returns a ChainLink at given position, oriented according to the given forward and up vectors.

   When entering play mode, you should see a chain constructed along the Bézier curve. You can edit the curve's control points and see it update interactively.