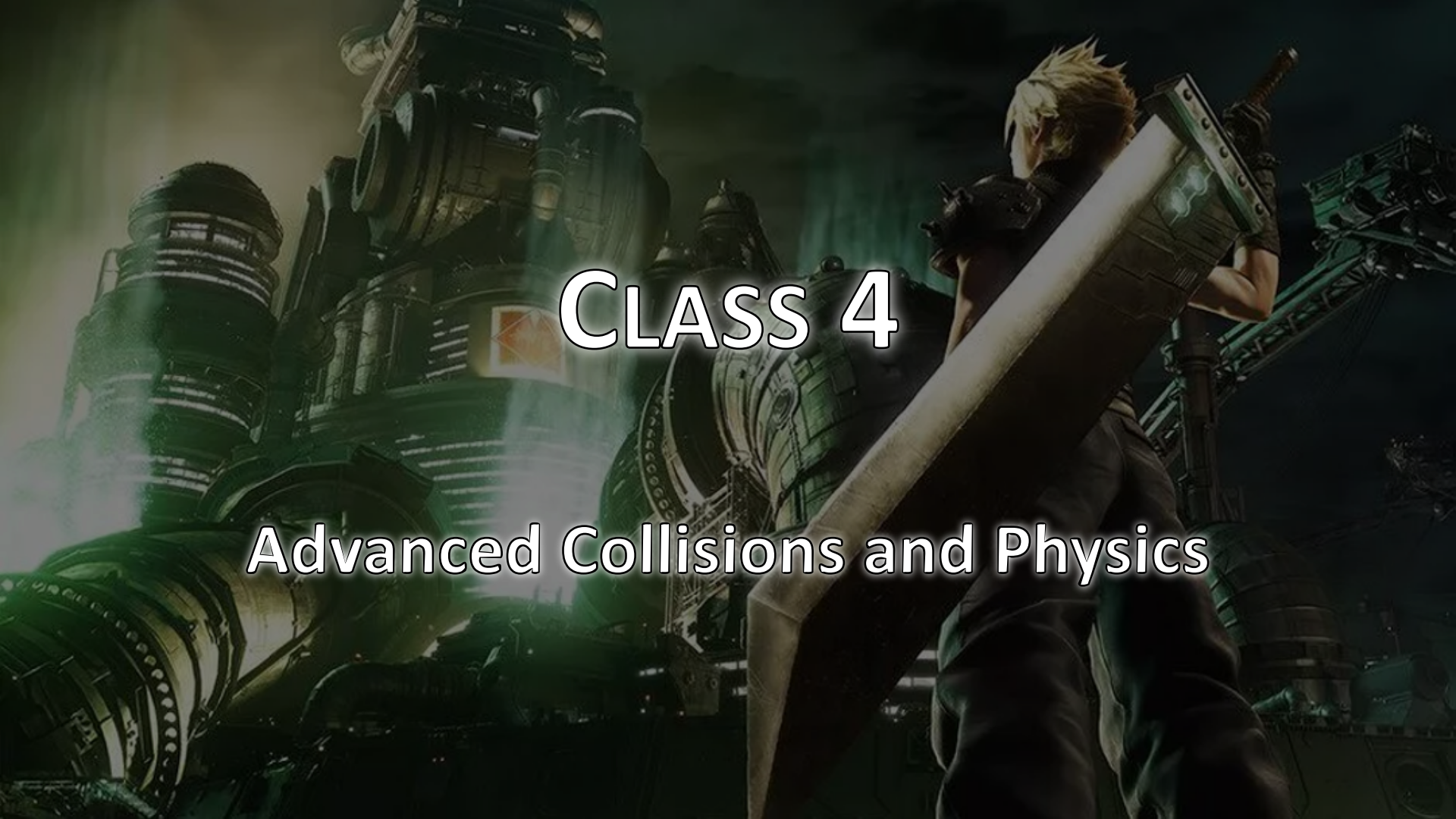
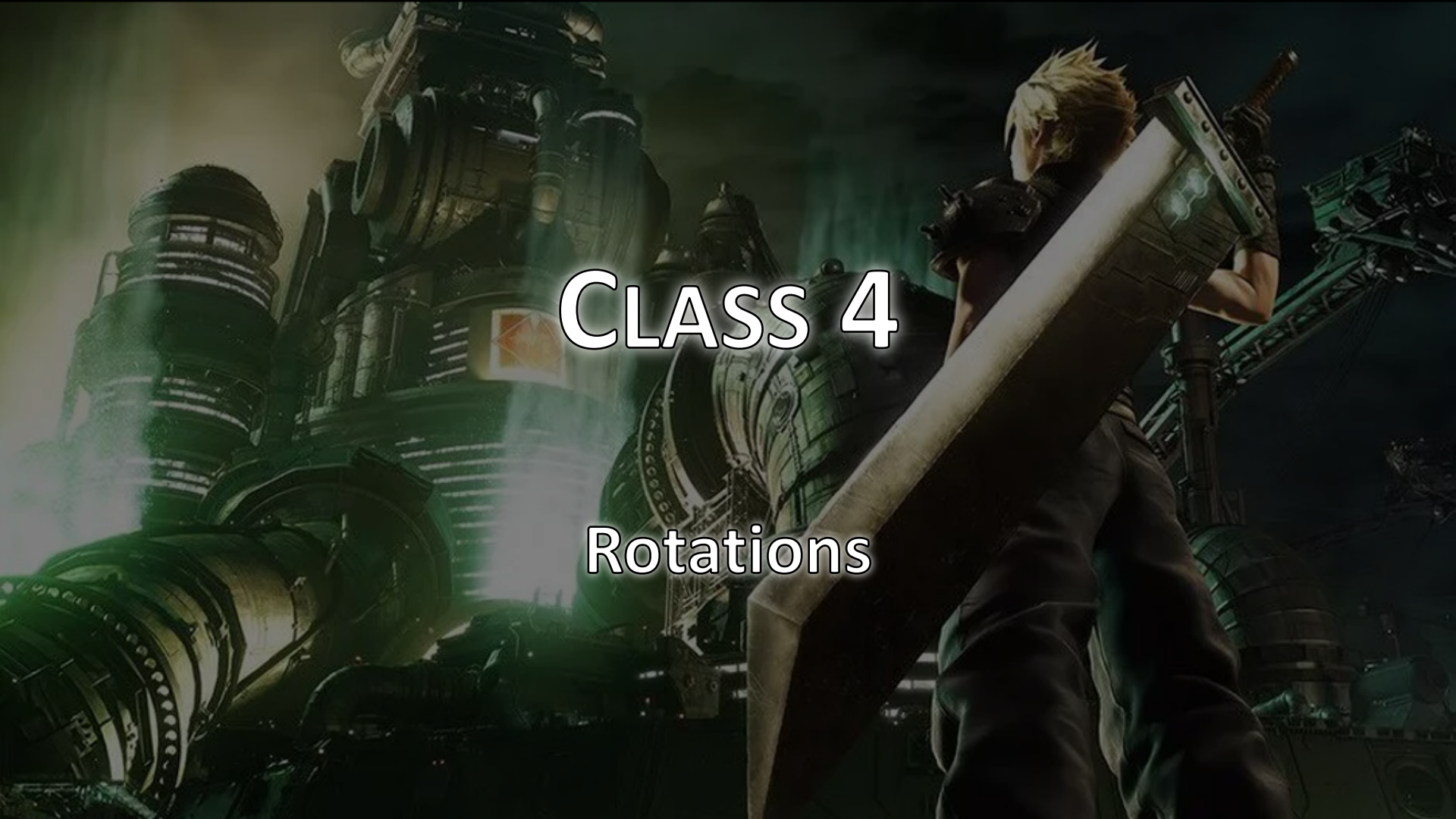

Class 4
Advanced Collisions and Physics

Collisions

- So far, we have had to write a new function every time we wanted to detect collisions between a new pair of shapes
- In this class, we will discuss a general method for detecting collisions that works for all convex shapes called the GJK algorithm
- We can easily replace our ellipsoid/triangle collisions with GJK collisions
- But first…


Class 4
Rotations

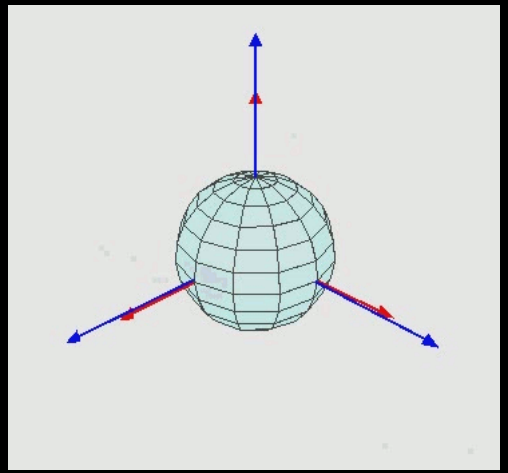
Rotations

- Rotations can be represented by Euler angles (i.e. roll, pitch, yaw), rotation matrices, or quaternions
- Useful fact: **any** arbitrary orientation can be achieved by a **single** rotation about some axis by some angle

Euler Angles

- When we talk about Euler angles, **we need to define an order of how the angles are set**
  - **We also need to say whether each angle in the sequence is measured in the object's transformed coordinate frame after each step (intrinsic), or the world coordinate frame (extrinsic angles)**

- Here, we have a sphere first rotating θ_1 about the z axis (up), then rotating θ_2 the x axis in **its transformed coordinate frame**, then rotating θ_3 the z axis in **its transformed coordinate frame**
- These are **intrinsic rotations**



- If the sphere first rotated θ_3 about the world z axis, then rotated θ_2 about the world x axis, and then rotated θ_1 about the world z axis, we would have **extrinsic rotations**
- We would also have the **same orientation as we did with the intrinsic rotations described to the left!**

Rotation Matrices

- Rotation matrices are 3x3 orthogonal matrices with determinant 1
  - This means that the columns of the matrix form an orthonormal basis of R^3 (a set of pairwise orthogonal and normalized vectors that span R^3)
- A rotation matrix says “x-axis, turn into my first column”, “y-axis, turn into my second column”, and “z-axis, turn into my third column”

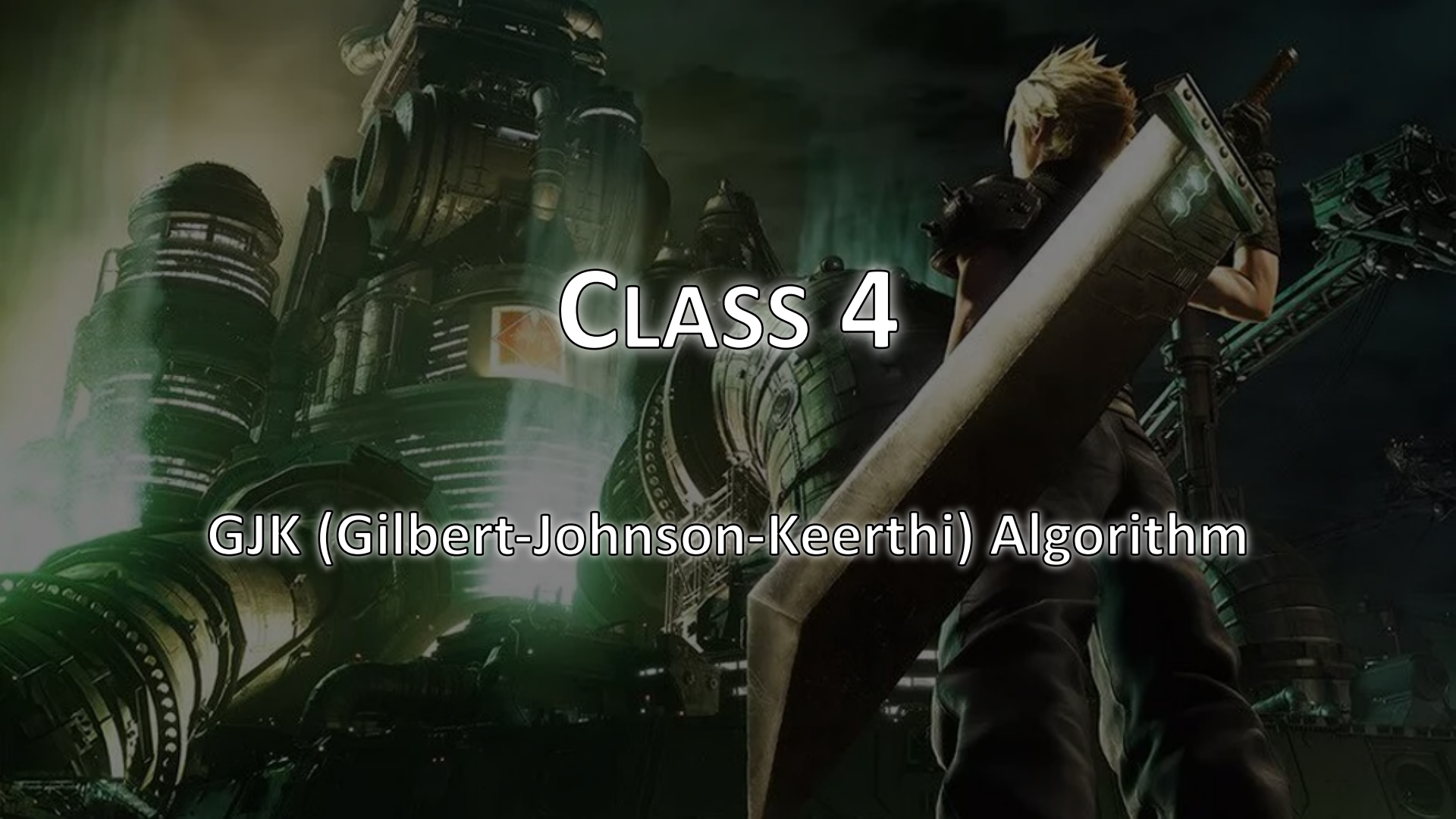
Quaternions

- Quaternions are a confusing subject, but it is helpful to know that there is a formula that takes in an axis of rotation u and an angle θ and gives you the quaternion corresponding to that rotation

$$\mathbf{q} = e^{\frac{\theta}{2}(u_x\mathbf{i}+u_y\mathbf{j}+u_z\mathbf{k})} = \cos\frac{\theta}{2} + (u_x\mathbf{i} + u_y\mathbf{j} + u_z\mathbf{k})\sin\frac{\theta}{2}$$

Rotations

- You will need to convert between rotation matrices and Euler angles in order to implement advanced collisions (if your transform component uses Euler angles)



Class 4

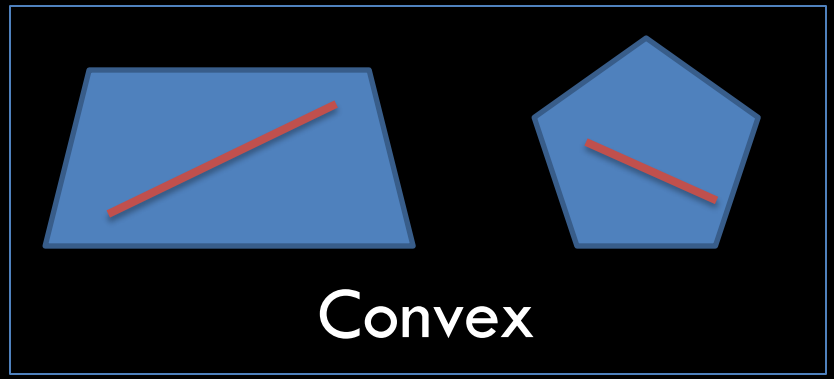
GJK (Gilbert-Johnson-Keerthi) Algorithm

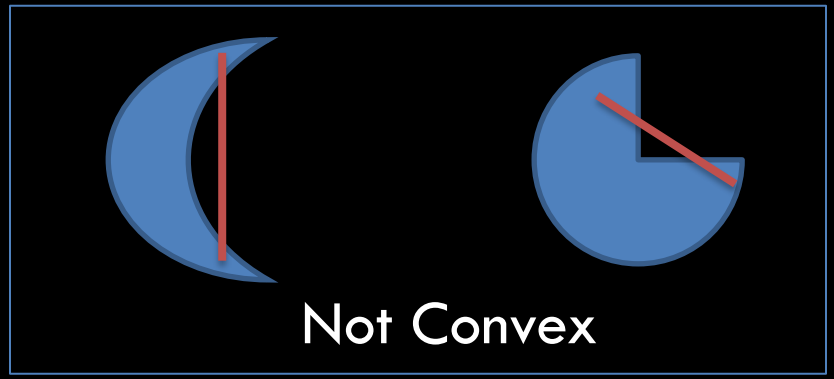
GJK Algorithm

SUPPORT FUNCTIONS

Convex Shapes

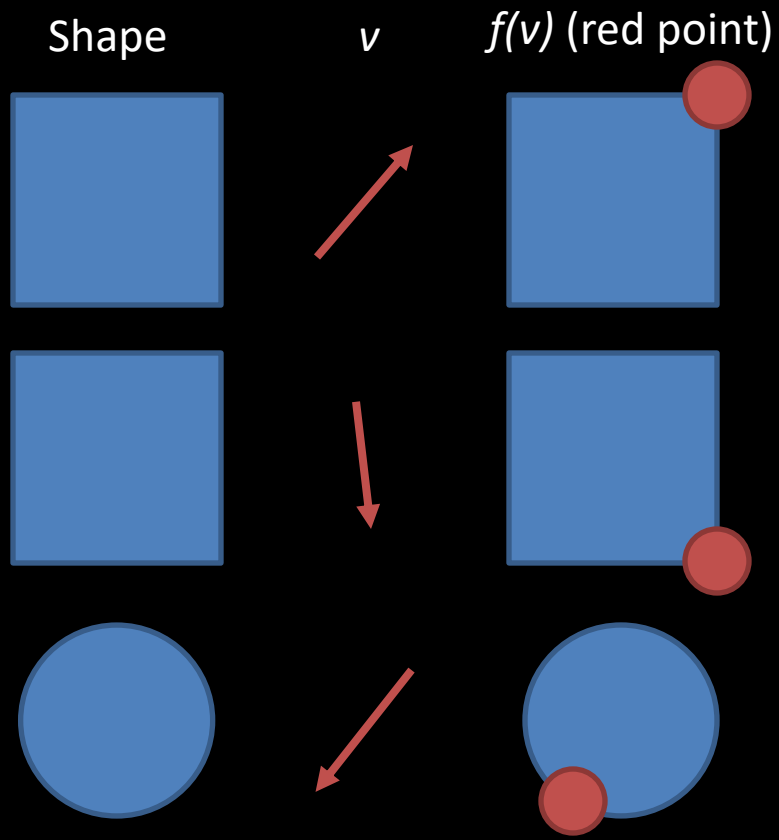
- A convex shape satisfies the requirement that the line segment connecting any point to any other point inside the shape exists entirely inside the shape





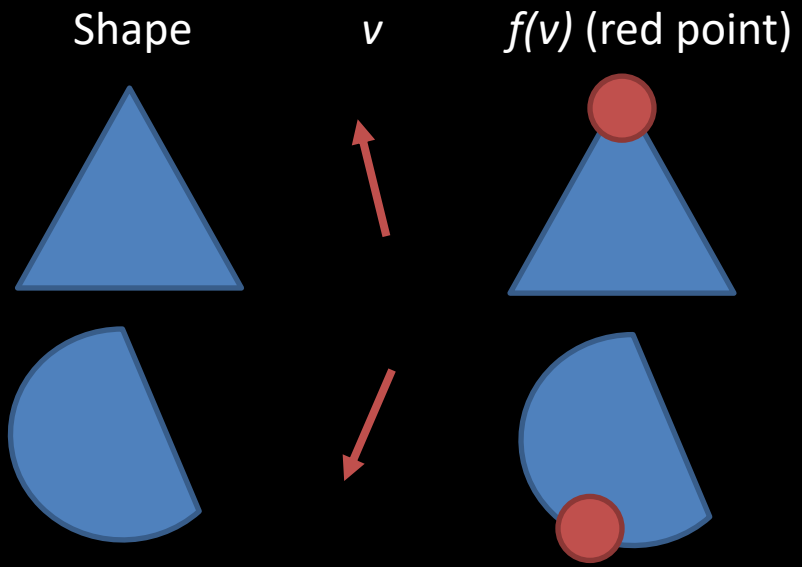
Support Functions

- We can define a **support function** for a convex shape
- A support function takes in a direction and then returns the point on the shape farthest in that direction
  - More precisely, given a direction v, the support function f of a shape s is $f(v) = \max_{\{p \in s\}} v \cdot p$
- Notice that the support function for the square always returns a corner
  - In the middle example, the vector is pointing slightly to the right, so the bottom right corner of the square maximizes the dot product



More Support Function Examples

- Given a direction v, the support function f of a shape s is $f(v) = \max_{\{p \in s\}} v \cdot p$



- Note the first example
  - The point returned is not necessary the exact point that the arrow is pointing to
  - Instead, the point returned is the point that maximizes the dot product

Support Functions Takeaway

- In a very broad sense, you can think of a support function as a function that **takes in a direction and returns a point on the boundary of the shape farthest in that direction**

Support Functions for 3D Shapes

Cone

A Cone primitive is a capped cone that is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cone with a radius of ρ at its base, and with its apex at $y = \eta$ and its base at $y = -\eta$. Then, the for the top angle α we have $\sin(\alpha) = \rho/\sqrt{\rho^2 + (2\eta)^2}$. Let $\sigma = \sqrt{x^2 + z^2}$, the distance from $(x, y, z)^{\mathrm{T}}$ to the y-axis. We choose as support mapping for A, the mapping

$$s_A((x, y, z)^{\mathrm{T}}) = \begin{cases} (0, \eta, 0)^{\mathrm{T}} & \text{if } y > \|(x, y, z)^{\mathrm{T}}\| \sin(\alpha) \\ (\frac{\rho}{\sigma}x, -\eta, \frac{\rho}{\sigma}z)^{\mathrm{T}} & \text{else, if } \sigma > 0 \\ (0, -\eta, 0)^{\mathrm{T}} & \text{otherwise.} \end{cases}$$

Cylinder

A Cylinder primitive is a capped cylinder that again is centered at the origin and whose central axis is aligned with the y-axis. Let A be a Cylinder with a radius of ρ, and with its top at $y = \eta$ and its bottom at $y = -\eta$. We find as support mapping for A the mapping

$$s_A((x, y, z)^{\mathrm{T}}) = \begin{cases} (\frac{\rho}{\sigma}x, \mathrm{sgn}(y)\eta, \frac{\rho}{\sigma}z)^{\mathrm{T}} & \text{if } \sigma > 0 \\ (0, \mathrm{sgn}(y)\eta, 0)^{\mathrm{T}} & \text{otherwise.} \end{cases}$$

Box

A Box primitive is a rectangular parallelepiped centered at the origin and aligned with the coordinate axes. Let A be a Box with extents $2\eta_x$, $2\eta_y$, and $2\eta_y$. Then, we take as support mapping for A,

$$s_A((x, y, z)^{\mathrm{T}}) = (\mathrm{sgn}(x)\eta_x, \mathrm{sgn}(y)\eta_y, \mathrm{sgn}(z)\eta_z)^{\mathrm{T}},$$

where $\mathrm{sgn}(x) = -1$, if $x < 0$, and 1, otherwise.

Sphere

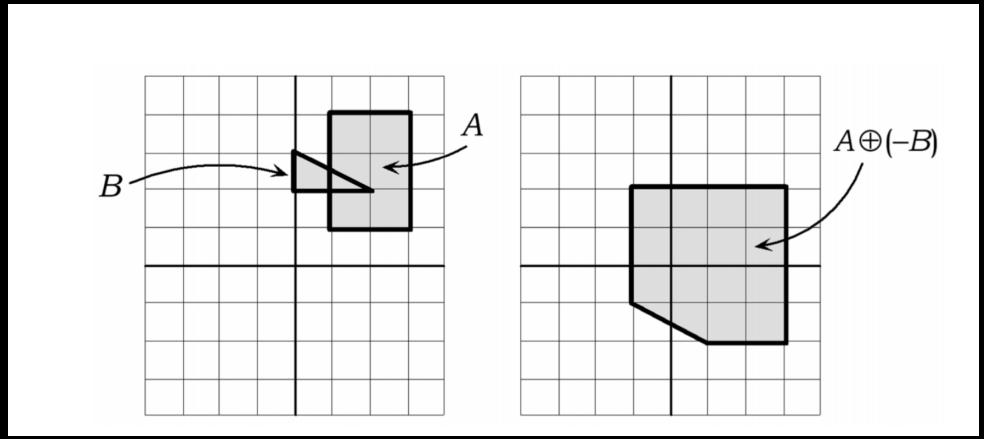
A Sphere primitive is a ball centered at the origin. The support mapping of a Sphere A with radius ρ is

$$s_A(\mathbf{v}) = \begin{cases} \frac{\rho}{\|\mathbf{v}\|}\mathbf{v} & \text{if } \mathbf{v} \neq \mathbf{0} \\ \mathbf{0} & \text{otherwise.} \end{cases}$$

From A Fast and Robust GJK Implementation for Collision Detection of Convex Objects
GINO VAN DEN BERGEN

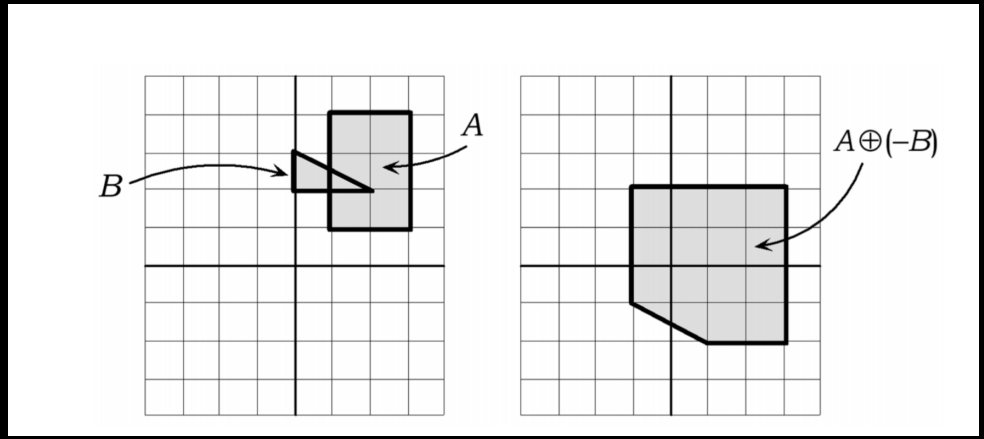
Minkowski Difference

- The Minkowski difference of shapes *A* and *B* is the set of points that are the difference of a point in A and a point in *B*
- In other words, if we subtract every point in *B* from every point in *A* we get the Minkowski difference



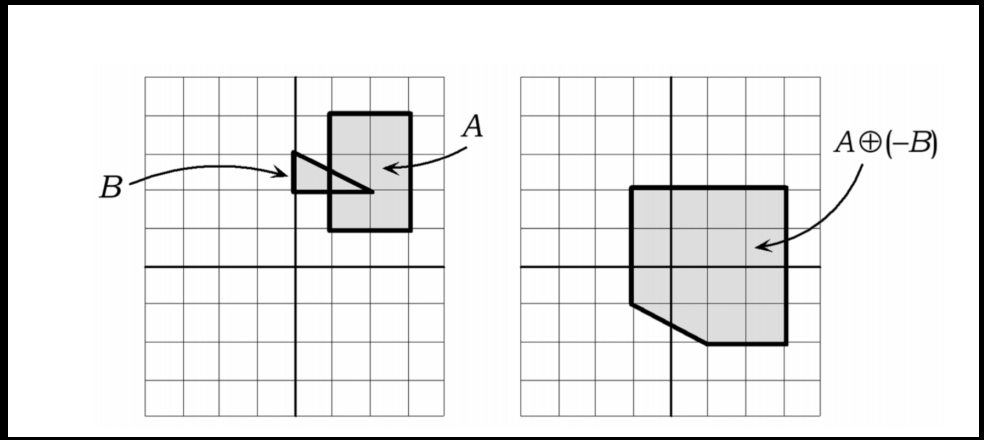
Minkowski Difference

- *A* and *B* are colliding if and only if the Minkowski difference contains the origin
  - Think about it this way: if *A* and *B* are colliding then they will overlap at some point in space
  - The Minkowski difference value resulting from this overlap is the origin
- We can use this fact to determine whether two shapes are colliding



Minkowski Difference and Support Functions

- It turns out that the Minkowski difference of two convex shapes is also convex
  - This means we can define the support function of the Minkowski difference!
- The support function f_{A-B} of the Minkowski difference of shapes A and B with support functions f_A and f_B is $f_{A-B}(v) = f_A(v) - f_B(-v)$
  - This identity requires a small proof that we will omit



GJK Algorithm

THE ALGORITHM

GJK Algorithm

- To figure out whether two shapes are colliding in 3D, we answer the question: “is the origin in the Minkowski difference?”
- To answer this question, we will try to create a **simplex** that contains the origin
  - A simplex is a generalization of a triangle to arbitrary dimensions
  - The 0-dimensional simplex is a point, the 1-dimensional simplex is a line, the 2-dimensional simplex is a triangle, and the 3-dimensional simplex is a tetrahedron
  - **The vertices of the simplex we are searching for are points returned by the support function of the Minkowski difference**

GJK Algorithm

- To be more concrete, **we have a collision if we can create a simplex consisting of vertices returned from the support function of the Minkowski difference** satisfying one of the following:
  - a 0-simplex (point) that contains the origin (the simplex itself is the origin)
  - a 1-simplex (line segment) that contains the origin (the origin is on the line segment)
  - a 2-simplex (triangle) that contains the origin (the origin exists on the triangle face)
  - A 3-simplex (tetrahedron) that contains the origin (the origin is inside the volume of the tetrahedron

GJK Algorithm Pseudocode

```
A = the newest vertex in the simplex
D = the next direction we plug into the Minkowski
difference support function
```

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

Pseudocode from
"Implementing GJK - 2006"
by Casey Muratori

do_simplex updates simplex and D

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

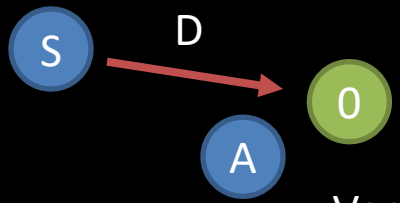
1. We first find an arbitrary point in the Minkowski difference (S)
2. Next, we find the point (A) in the Minkowski difference farthest in the opposite direction of S
3. If dot(A, D) < 0, then we did not pass the origin when we walked from vertex S to vertex A

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

1. We first find an arbitrary point in the Minkowski difference (S)
2. Next, we find the point (A) in the Minkowski difference farthest in the opposite direction of S
3. If dot(A, D) < 0, then we did not pass the origin when we walked from vertex S to vertex A



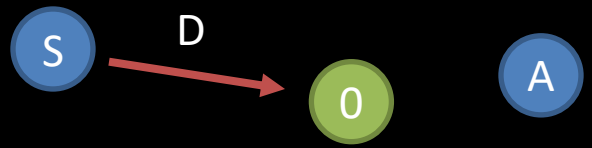
Vertex A did not pass the origin

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

1. We first find an arbitrary point in the Minkowski difference (S)
2. Next, we find the point (A) in the Minkowski difference farthest in the opposite direction of S
3. If dot(A, D) < 0, then we did not pass the origin when we walked from vertex S to vertex A



Vertex A passed the origin

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

1. We first find an arbitrary point in the Minkowski difference (S)
2. Next, we find the point (A) in the Minkowski difference farthest in the opposite direction of S
3. If dot(A, D) < 0, then we did not pass the origin when we walked from vertex S to vertex A
4. The Minkowski difference is **convex**, so if we did not pass the origin when we walked from vertex S to vertex A, then we know that there is no collision

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

5. If we succeeded in passing the origin, then we add vertex A to the simplex
6. Next, we run the do_simplex function, which decides if we are done. If not, the function finds the next direction D that we should plug into the support function. The support function will then give us the next vertex we will add to the simplex.

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

- We can see that the main GJK algorithm is not too complicated
- The function do_simplex is doing most of the heavy lifting here
- do_simplex does the following:
  - Decide if the current simplex contains the origin (if so, GJK returns true and the simplex)
  - If the current simplex does not contain the origin, find the optimal direction to search and update simplex and D accordingly

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

```
bool do_simplex(simplex, D):
  if contains_origin(simplex): return true
  return handle_simplex(simplex, D)
```

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

GJK Algorithm Pseudocode

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

```
bool do_simplex(simplex, D):
  if contains_origin(simplex): return true
  return handle_simplex(simplex, D)
```

- When we call handle_simplex, the simplex will have 2, 3, or 4 vertices
- We need to figure out the optimal way to change the simplex in each case
- The optimal way to change the simplex will capture the origin in a simplex in the minimum number of iterations

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

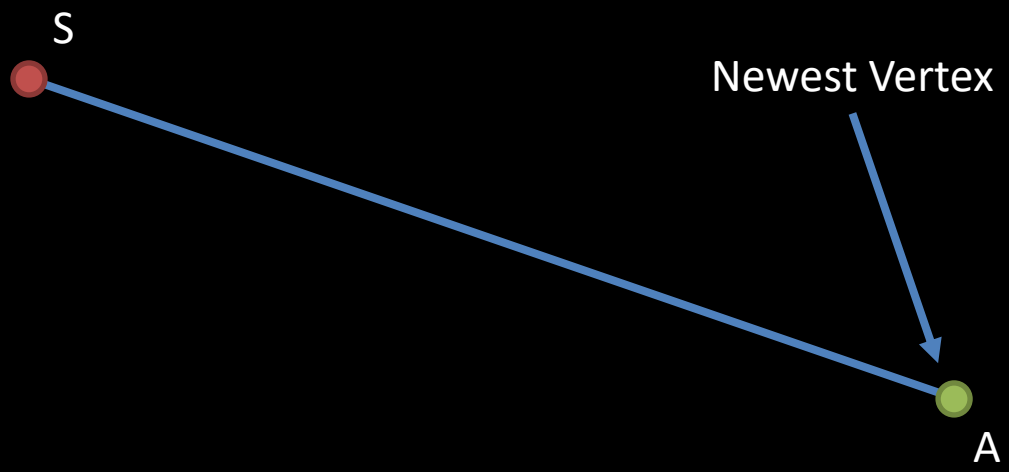
Updating the Simplex

- The content here is adapted from “Implementing GJK - 2006”, a video by Casey Muratori
- This video points out the important insight that **the position of the vertex that was last added to the simplex gives us hints about which direction we should plug into the Minkowski support function in the next iteration** (to get the next vertex of the simplex)
- In the following slides, the vertex that was last added to the simplex will be called *A*

Pseudocode from
“Implementing GJK - 2006”
by Casey Muratori

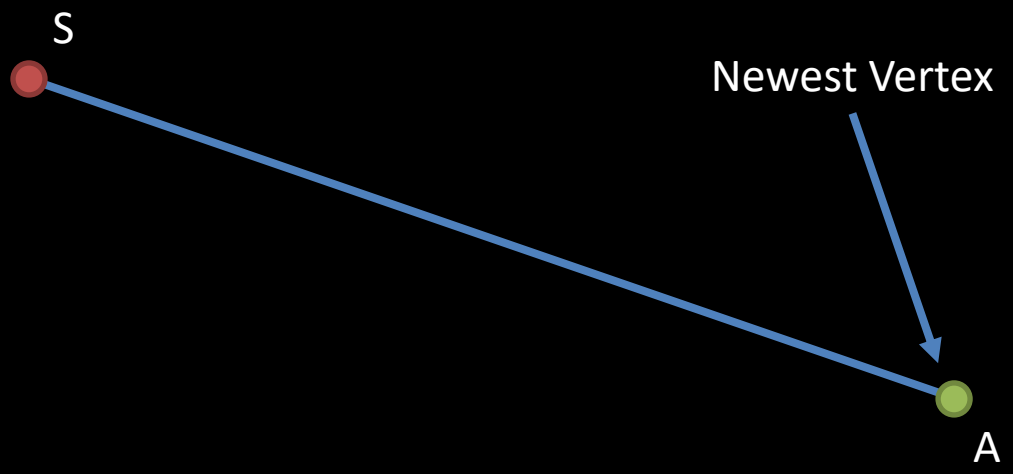
Handling the 1-Simplex (Line)

- Remember that every vertex in the simplex is a point on the boundary of the Minkowski difference
- We need to find a new direction to plug into the support function of the Minkowski difference



Handling the 1-Simplex (Line)

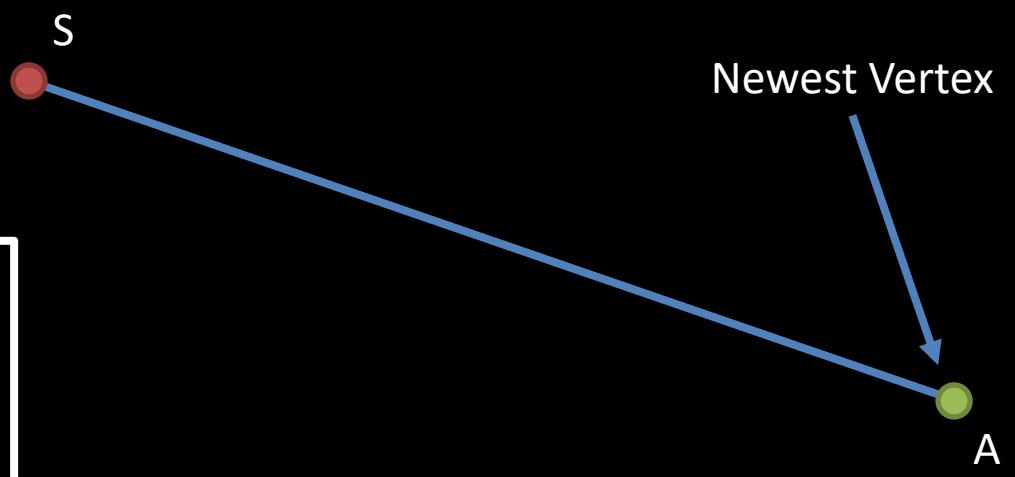
- How did the simplex become a line?



Handling the 1-Simplex (Line)

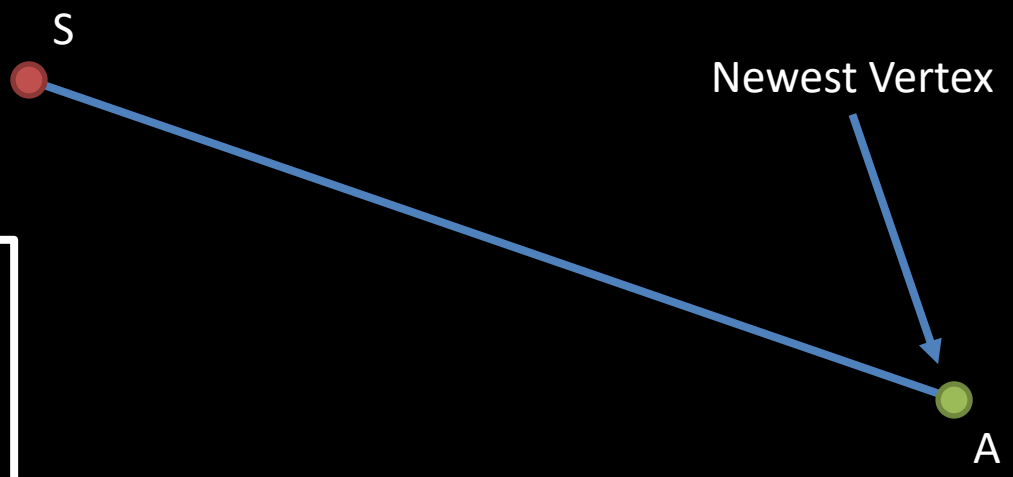
- Let's look at the pseudocode again…

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```



Handling the 1-Simplex (Line)

- First, we add some point (S) to simplex

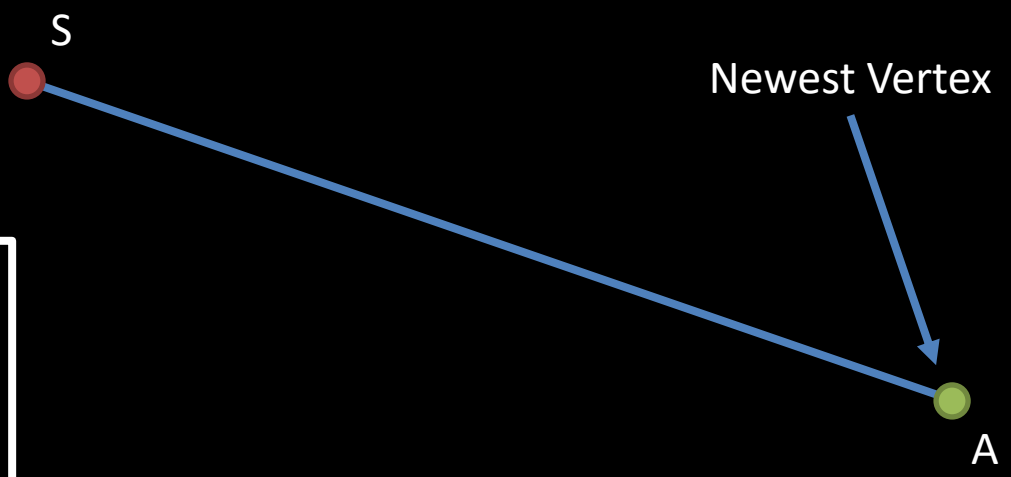



```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

Handling the 1-Simplex (Line)

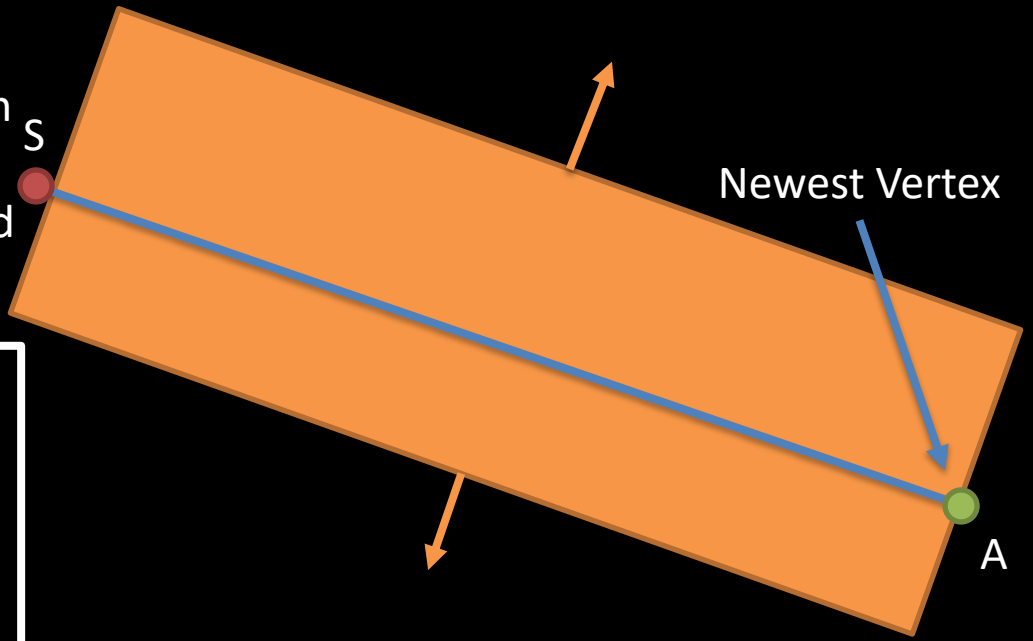
- Then, we plug –S into the support function
- Remember that we only got to this point in the algorithm because we passed the origin when we walked from S to A

```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```



Handling the 1-Simplex (Line)

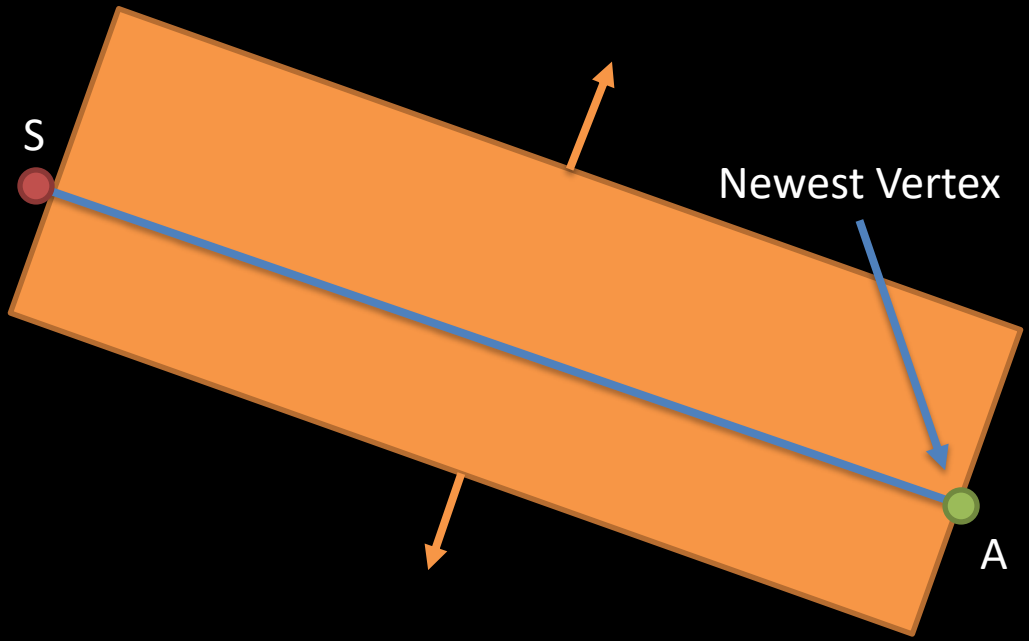
- Since we know that we passed the origin, we know that the origin exists somewhere in the orange area between S and A (extending out to infinity)
- The origin cannot be behind S or behind A




```
pair<bool, simplex> gjk(support_func):
  S = support_func(arbitrary_direction)
  simplex = [S]
  D = -S
  while True:
    A = support_func(D)
    if dot(A, D) < 0: return (false, [])
    simplex.append(A)
    if do_simplex(simplex, D): return (simplex, D)
```

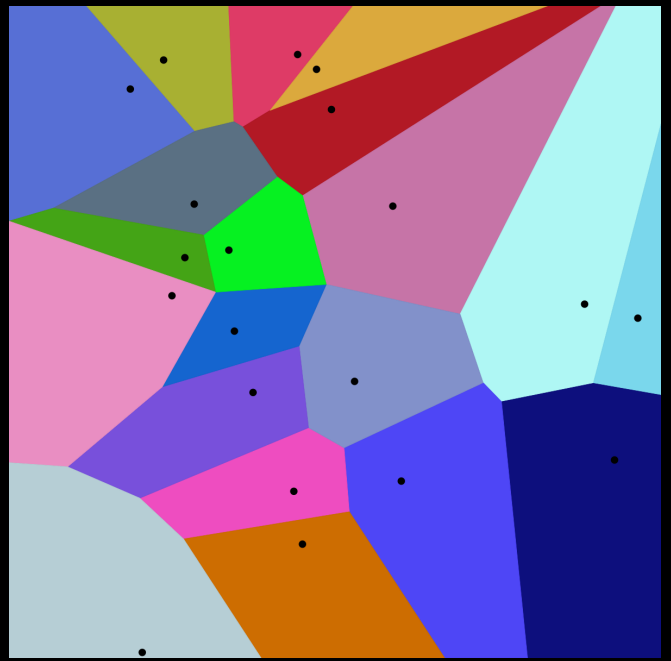
Handling the 1-Simplex (Line)

- So, the new direction D is the vector perpendicular to the line segment SA in the direction of the origin!
- We don't need to adjust the simplex at all in this case before adding support_function(D) to the
- The 1-simplex case is solved!



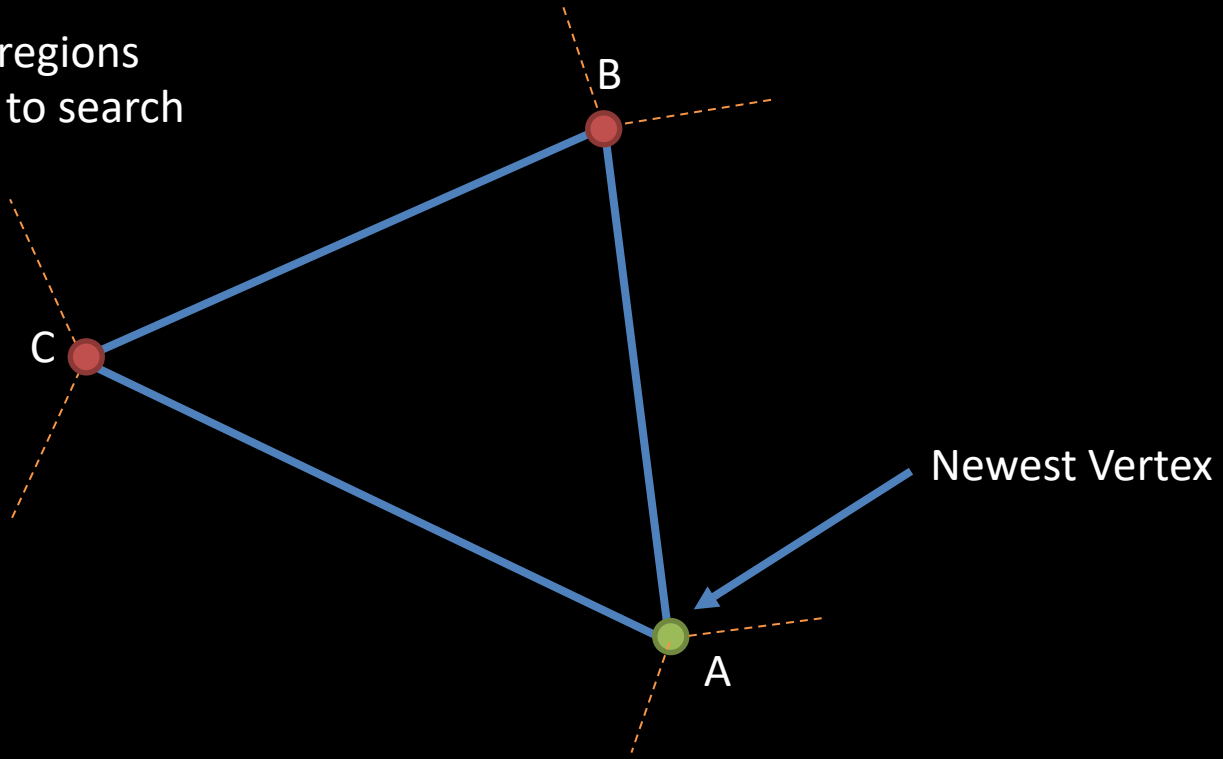
Aside: Voronoi Diagrams

- A Voronoi diagram shows a partition of the plane where the ”Voronoi region” of a seed *s* (a black dot in the figure) is the set of points that are closer to *s* than any other seed
- It is useful to think about this type of diagram when dealing with the 2-simplex case



Handling the 2-Simplex (Triangle)

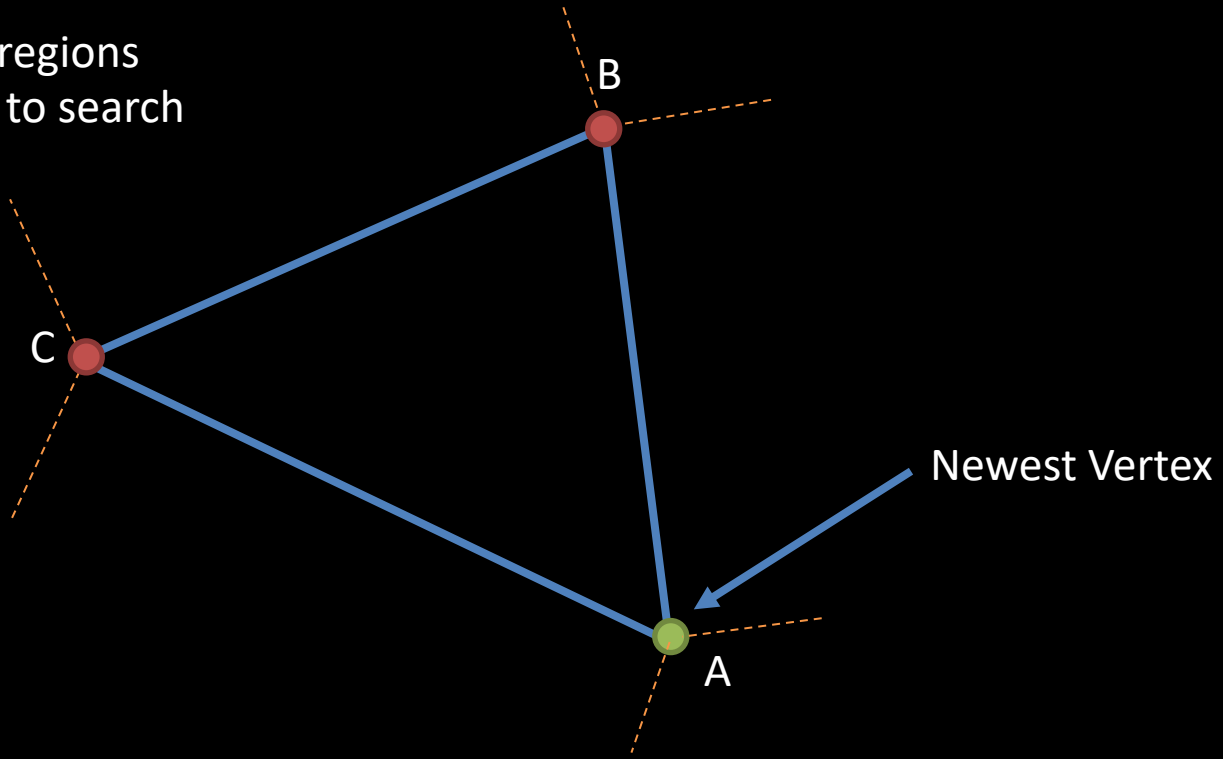
- There are 8 (Voronoi) regions where we might want to search for the origin



Handling the 2-Simplex (Triangle)

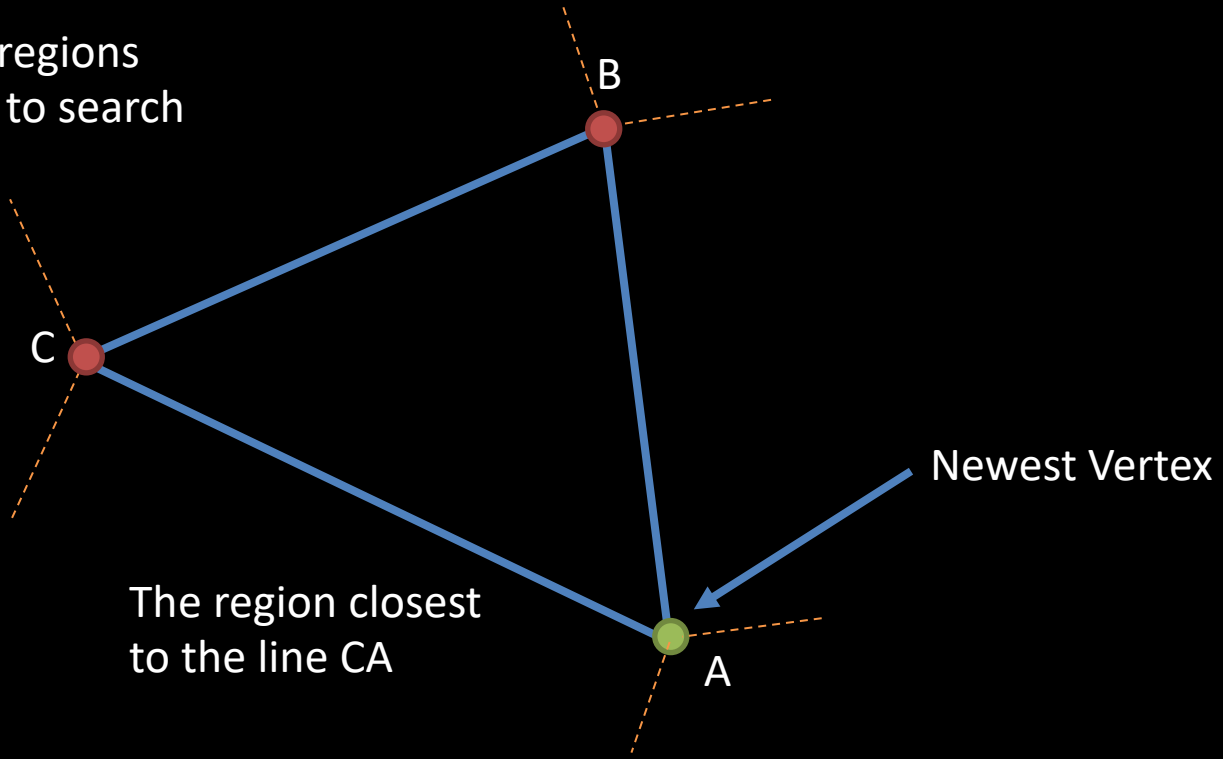
- There are 8 (Voronoi) regions where we might want to search for the origin

The region closest to the vertex C



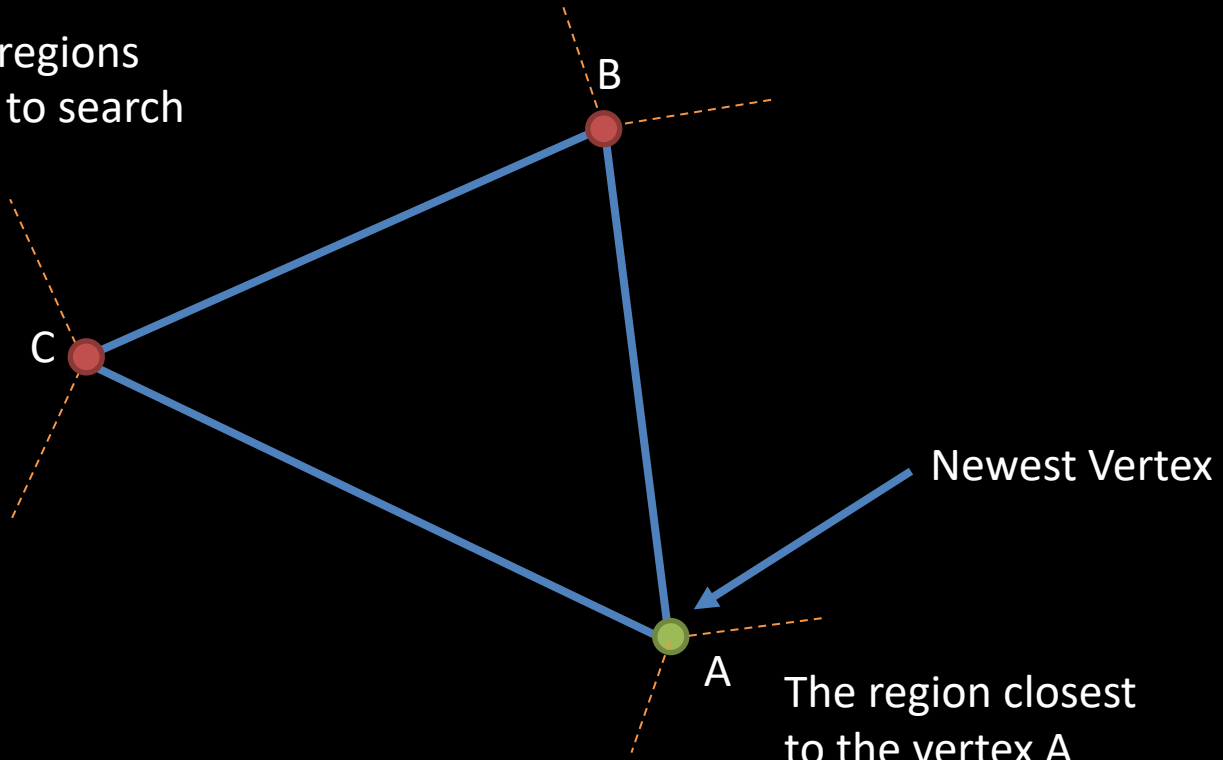
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



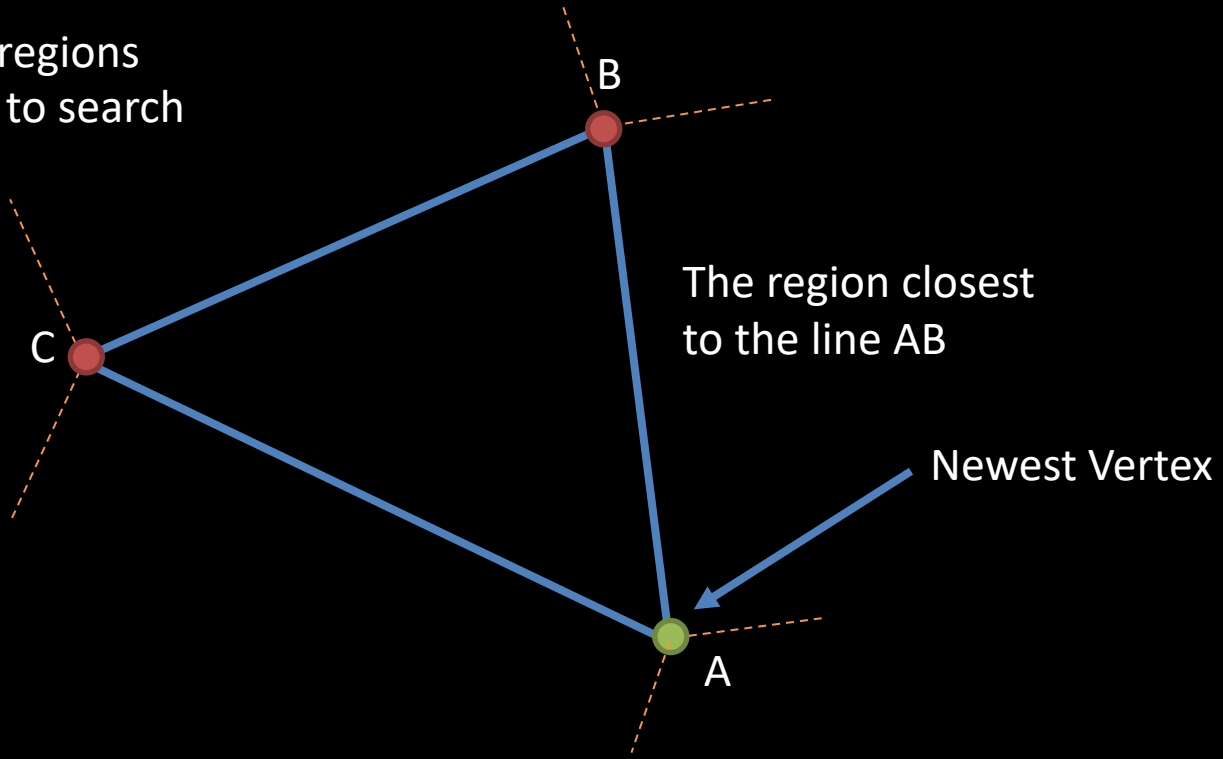
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



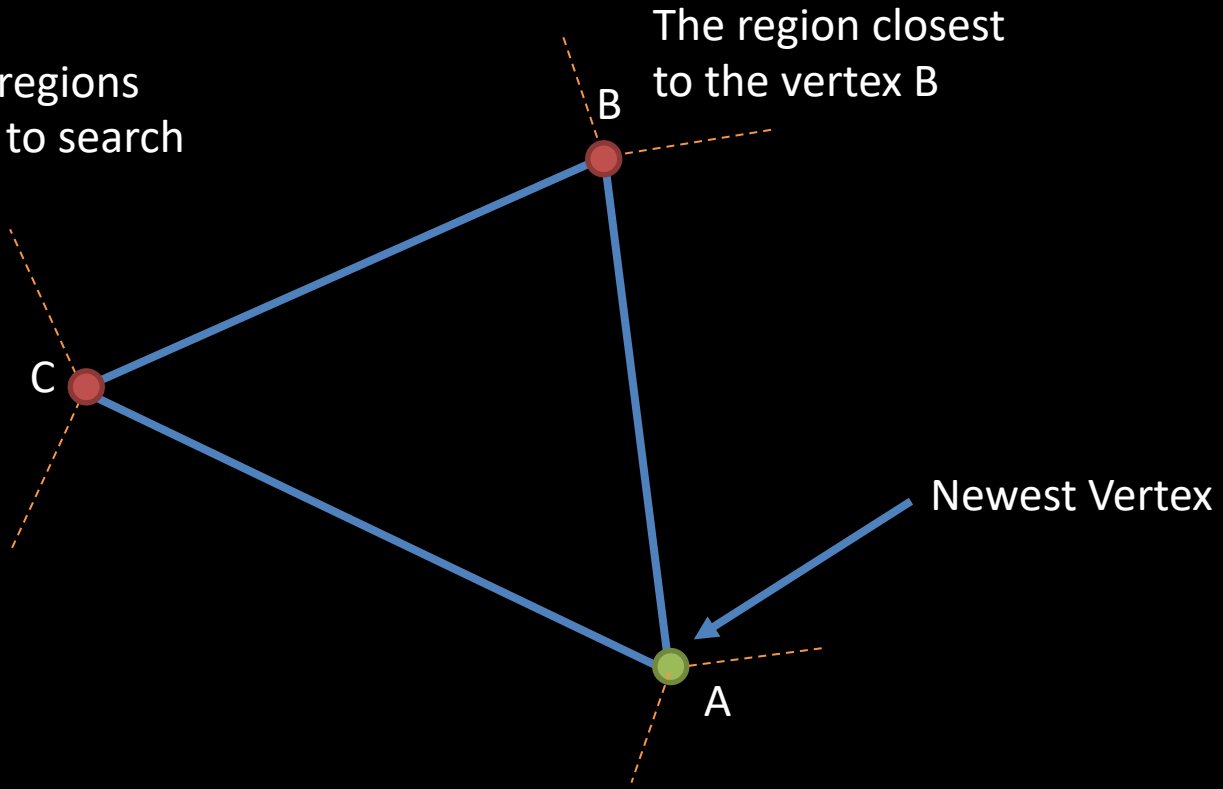
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



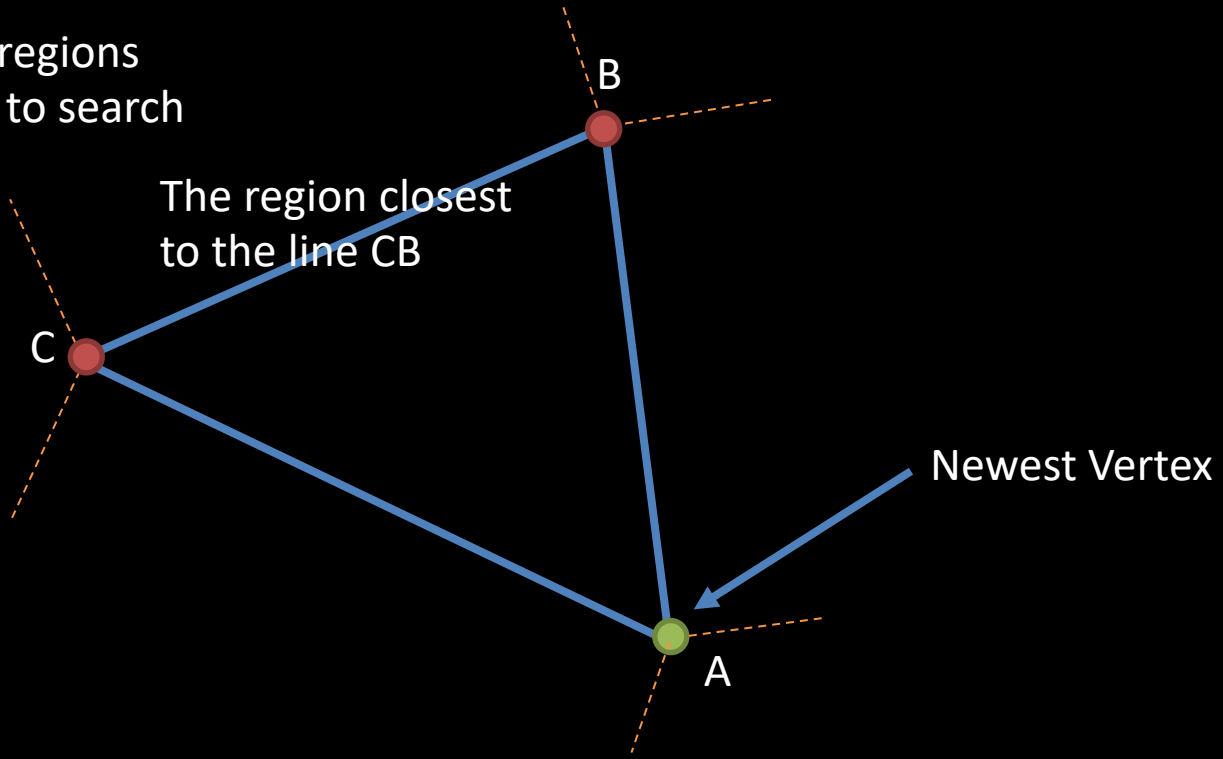
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



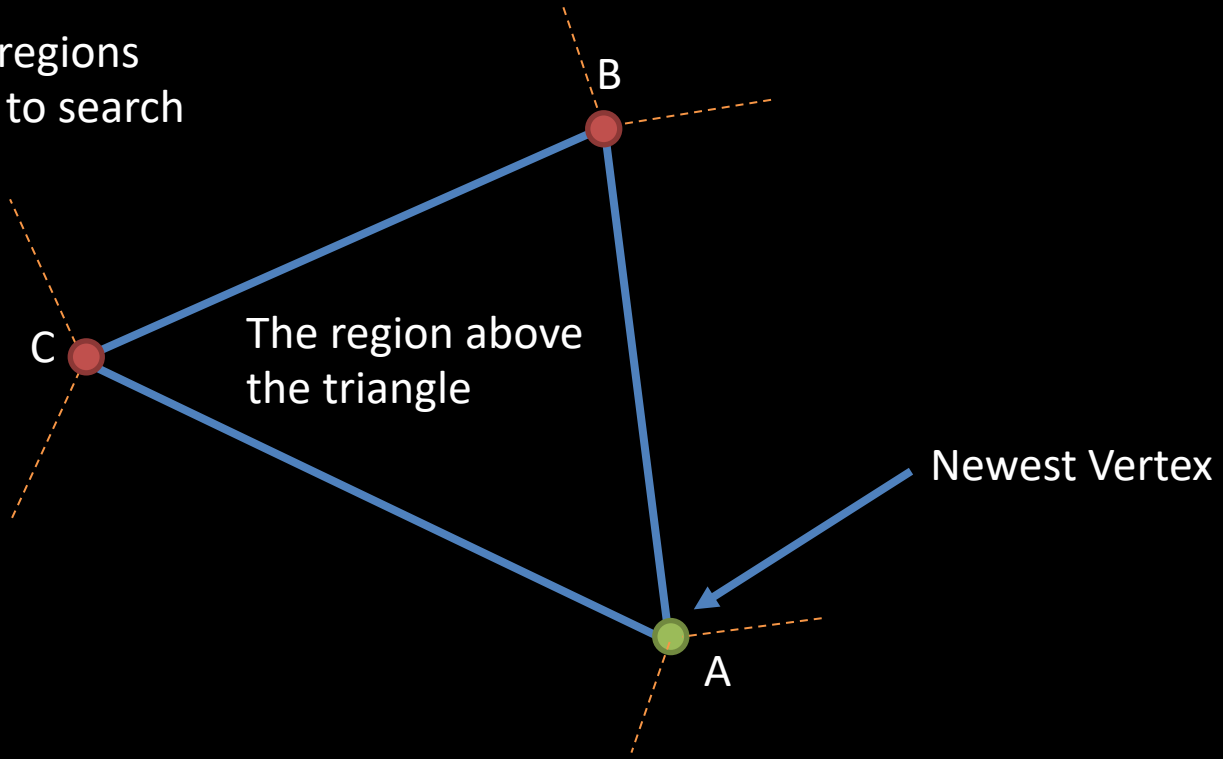
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



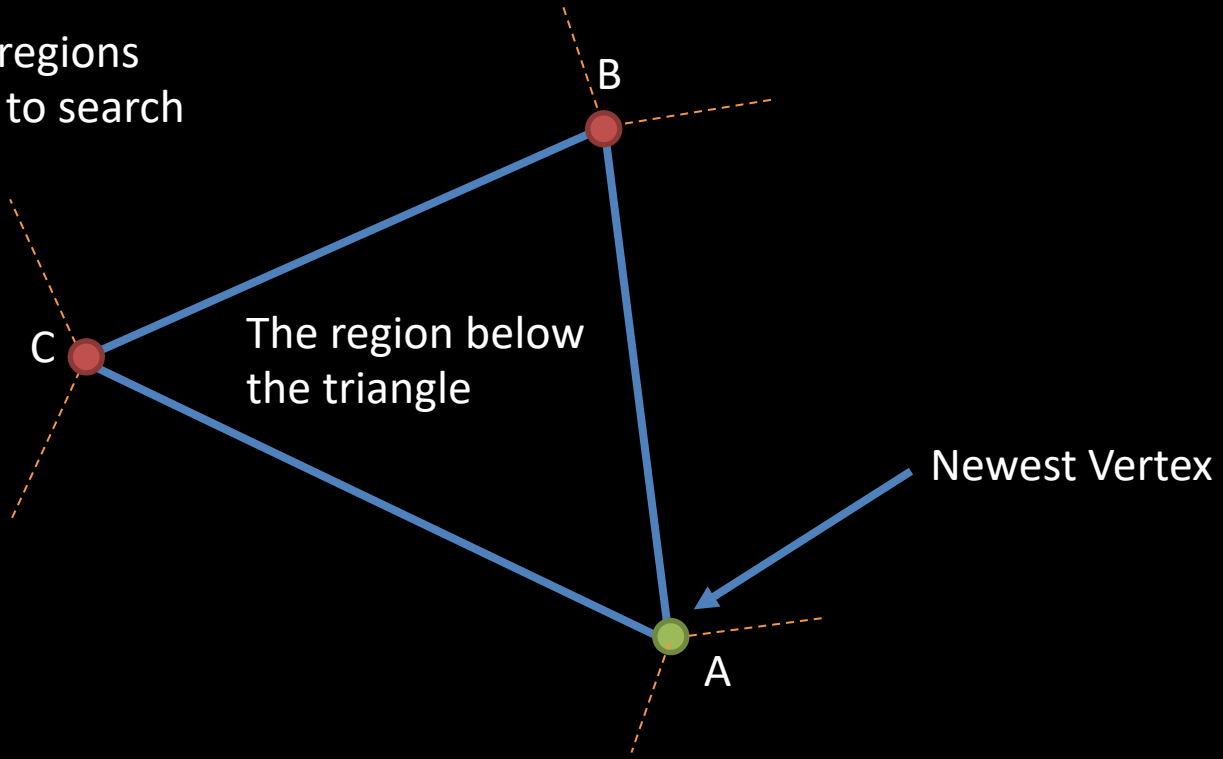
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



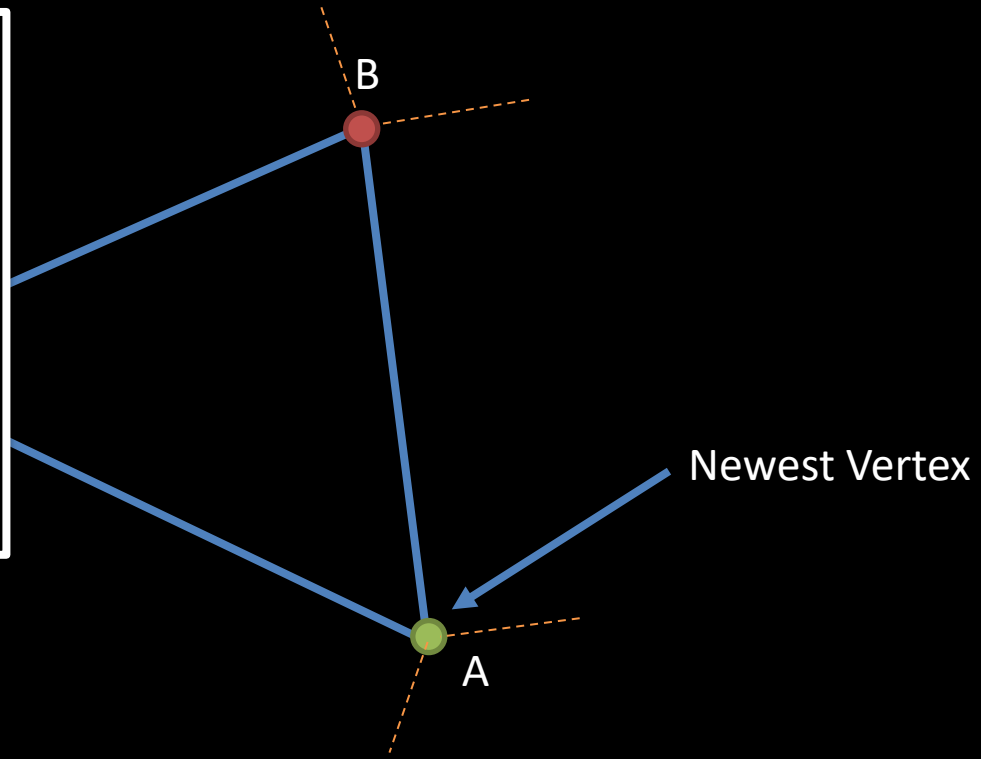
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin



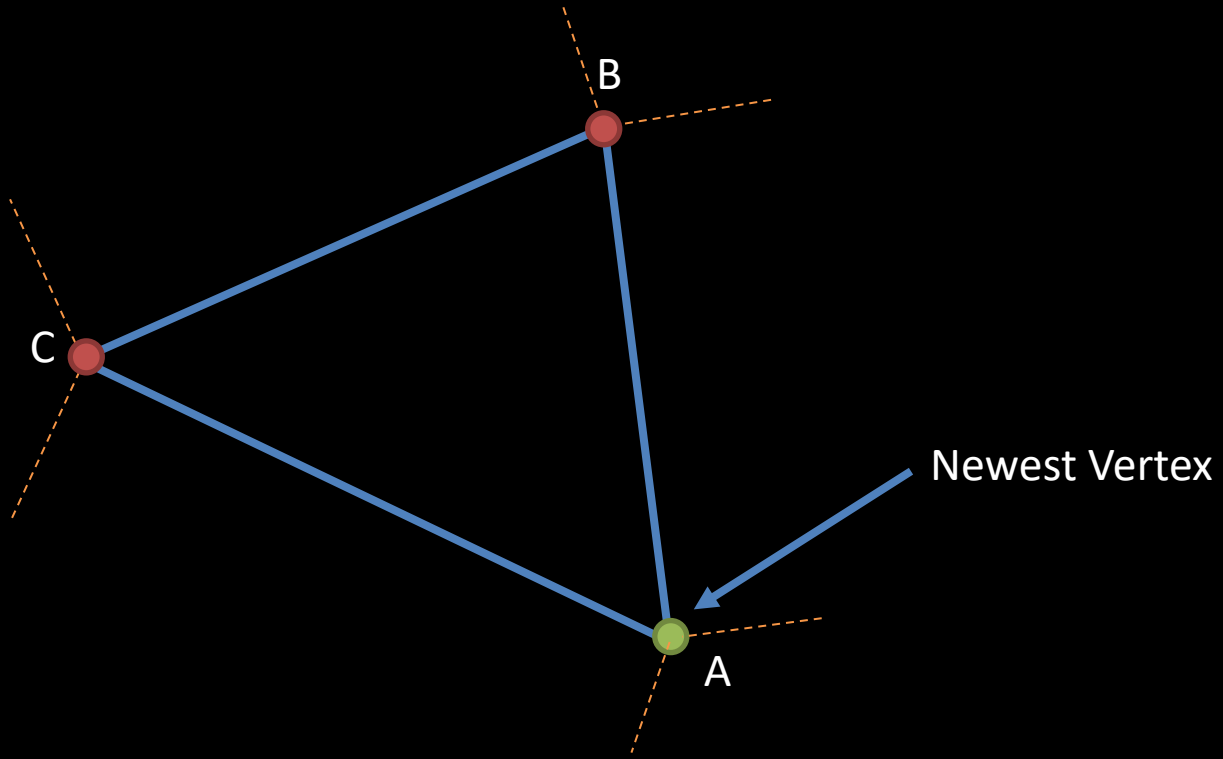
Handling the 2-Simplex (Triangle)

- There are 8 (Voronoi) regions where we might want to search for the origin
- Keep in mind that, if this triangle were on the XY plane, then these regions would extend straight forward and backward in the z-direction



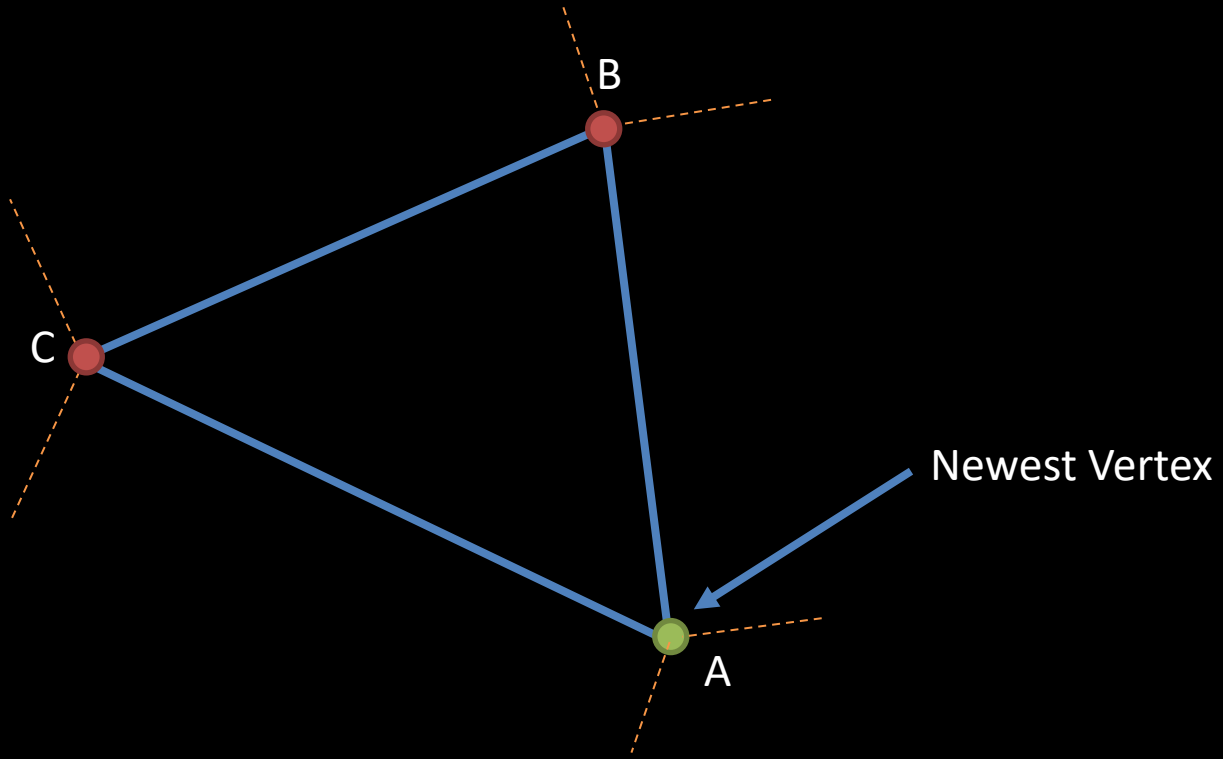
Handling the 2-Simplex (Triangle)

- Our task is to find the region containing the origin
- When we find that region, we will send our new direction D into that region!



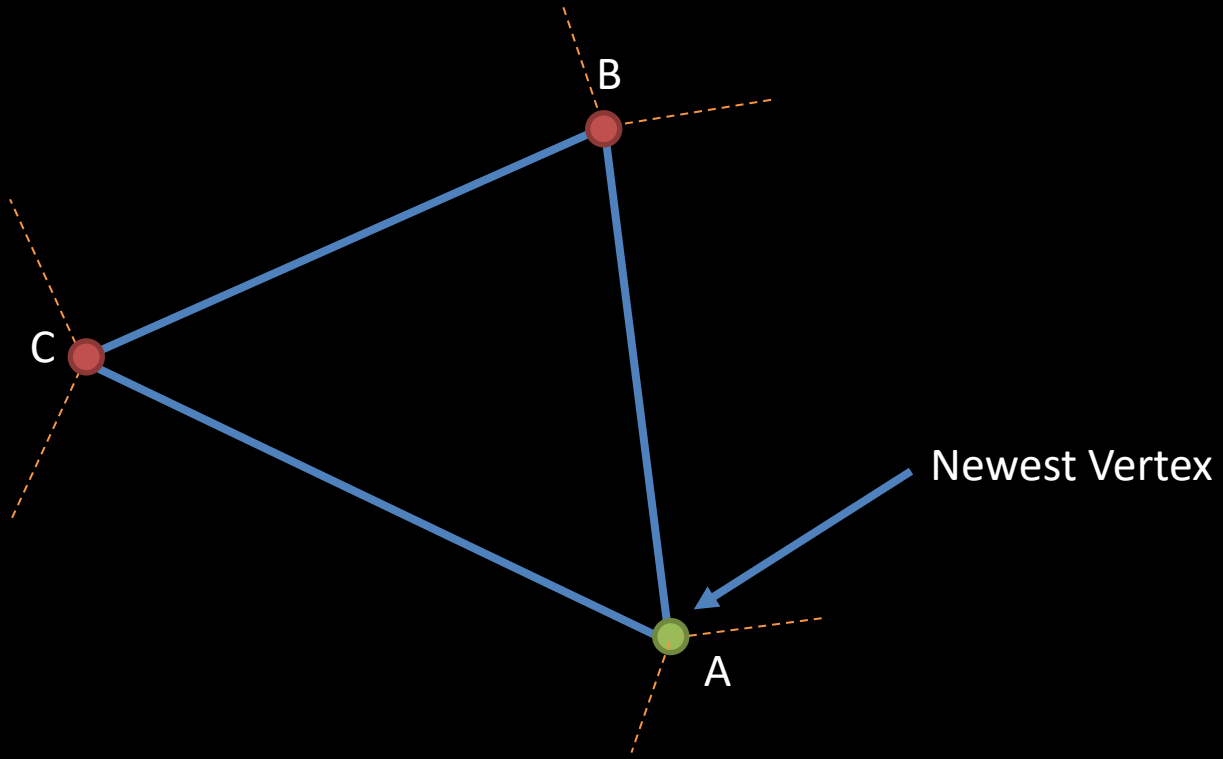
Handling the 2-Simplex (Triangle)

- But we don’t have to check **every** region!
- We have ruled out some of these regions in previous iterations!

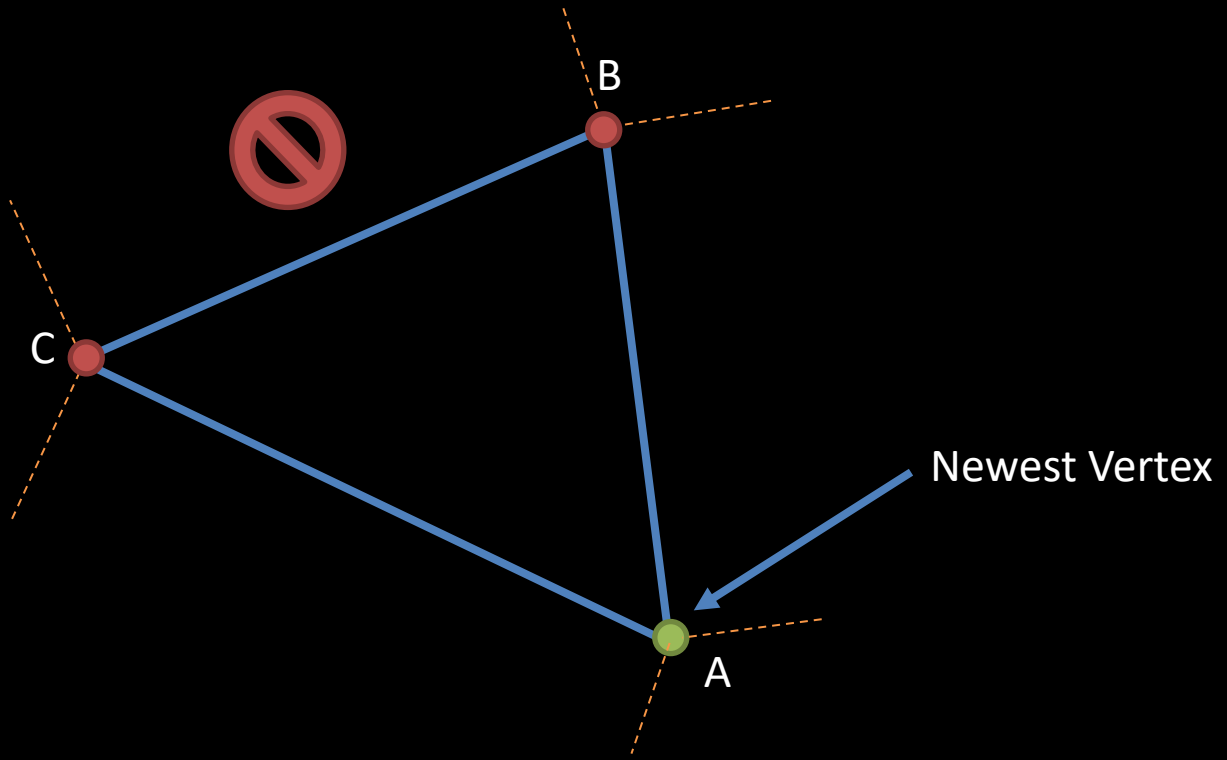


Handling the 2-Simplex (Triangle)

- Remember that the newest vertex is *A*
- When our simplex was CB, we decided the origin was in the direction of A
- Therefore, the CB region does not contain the origin

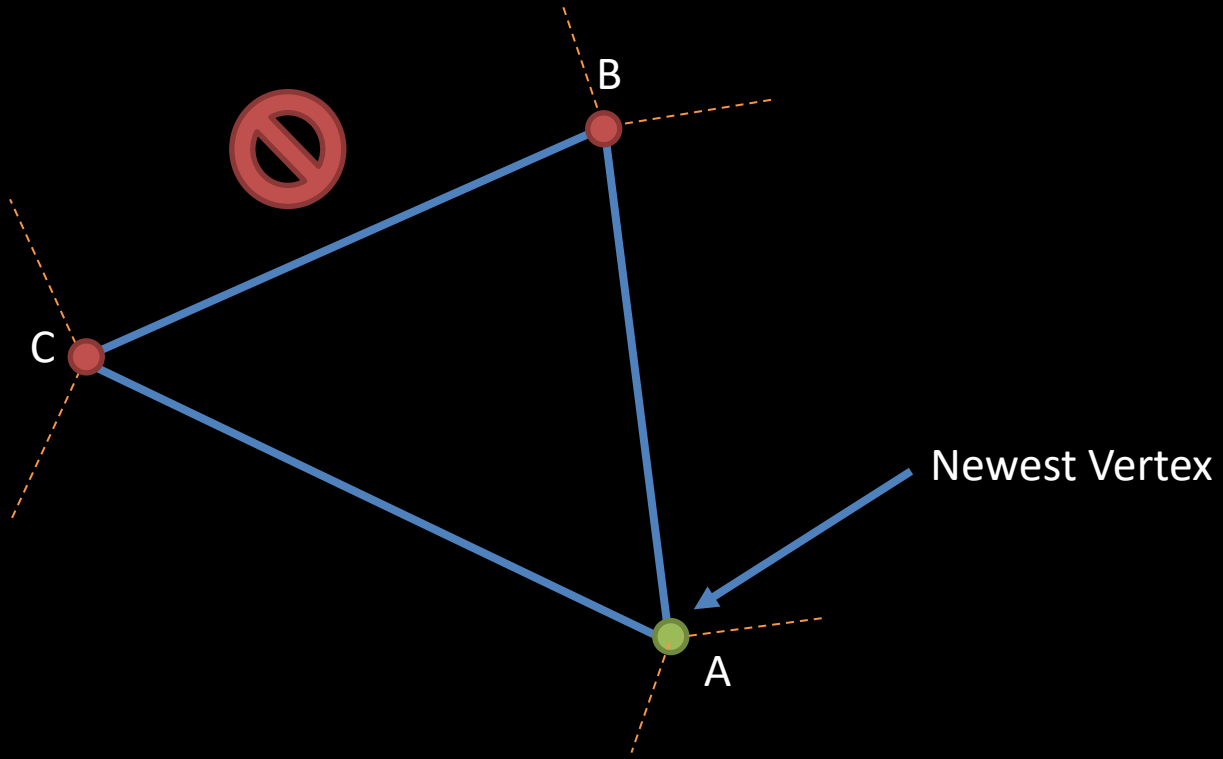


Handling the 2-Simplex (Triangle)



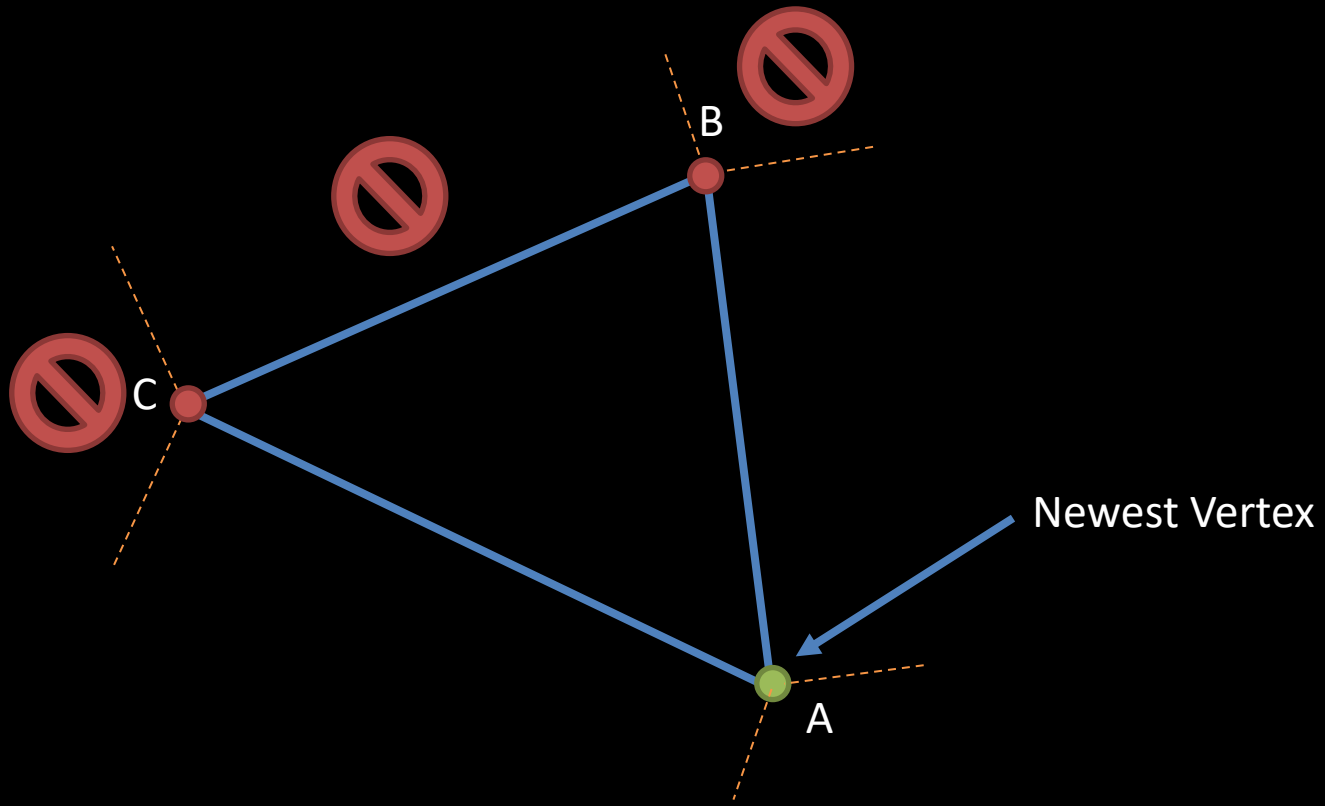
Handling the 2-Simplex (Triangle)

- The C and B regions also do not contain the origin!
- Consider the 1-simplex case to see why!



Handling the 2-Simplex (Triangle)

- So, we have 5 regions to check: CA, A, AB, above the triangle, and below the triangle!
- We use a bunch of dot and cross products to decide which region the origin is in



2-Simplex Pseudocode

```
simplex2_case(simplex, D):
  A = simplex[2] // newest vertex in the simplex
  B = simplex[1]
  C = simplex[0]

  // let's arbitrarily say this normal points "above" the triangle (in the previous slides, this normal would point at us)
  // this way, we don't have to maintain a winding order
  normal = cross(B-A,C-A)

  if dot(cross(normal, C-A), -A) > 0: // true if origin is in CA region or A region
    if dot(C-A, -A) > 0: // true if origin is in CA region
      simplex = [C,A]
      D = cross(cross(C-A, -A), C-A)
      return
    else: // executes if origin is in A region
      simplex = [A]
      D = -A
      return
  else:
    if dot(cross(B-A, normal), -A) > 0: // true if the origin is in BA region or A region
      if dot(B-A, -A) > 0: // true if origin is in BA region
        simplex = [B,A]
        D = cross(cross(B-A, -A), B-A)
        return
      else: // executes if origin is in A region
        simplex = [A]
        D = -A
        return
    else: // executes if origin is above or below the triangle
      if dot(normal, -A) > 0: // true if origin is above triangle
        simplex = [A,B,C]
        D = normal
        return
      else: // executes if origin is below triangle
        simplex = [A,B,C]
        D =  -normal
        return
```

Pseudocode from "Implementing GJK - 2006" by Casey Muratori

Handling the 3-Simplex

- Remember that we want a simplex with at most 4 vertices!
- When we have a 3-simplex (tetrahedron), we just reduce the problem back to the 2-simplex case
  - Given a tetrahedron where vertex *A* is the newest vertex, we calculate which of the planes of faces ABC, ABD, and ACD is closest to the origin
  - Make sure the closest plane is actually facing the origin
  - Then we just run our 2-simplex code on that face!

Verifying the Simplex

- We know we are done running GJK if we fail to pass the origin or if the simplex contains the origin
- A 2-simplex contains the origin if the origin exists on the face of the triangle
  - We should first verify whether the origin exists in the plane of the triangle
  - If the origin exists in the plane of the triangle, then the simplex contains the origin if, when we take any edge of the simplex, the origin is on the same side of the edge as the vertex of the simplex opposite the edge
- A 3-simplex contains the origin if, when we take any face of the simplex, the origin is on the same side of the face as the vertex of the simplex opposite the face

Evaluating the Minkowski Difference Support function

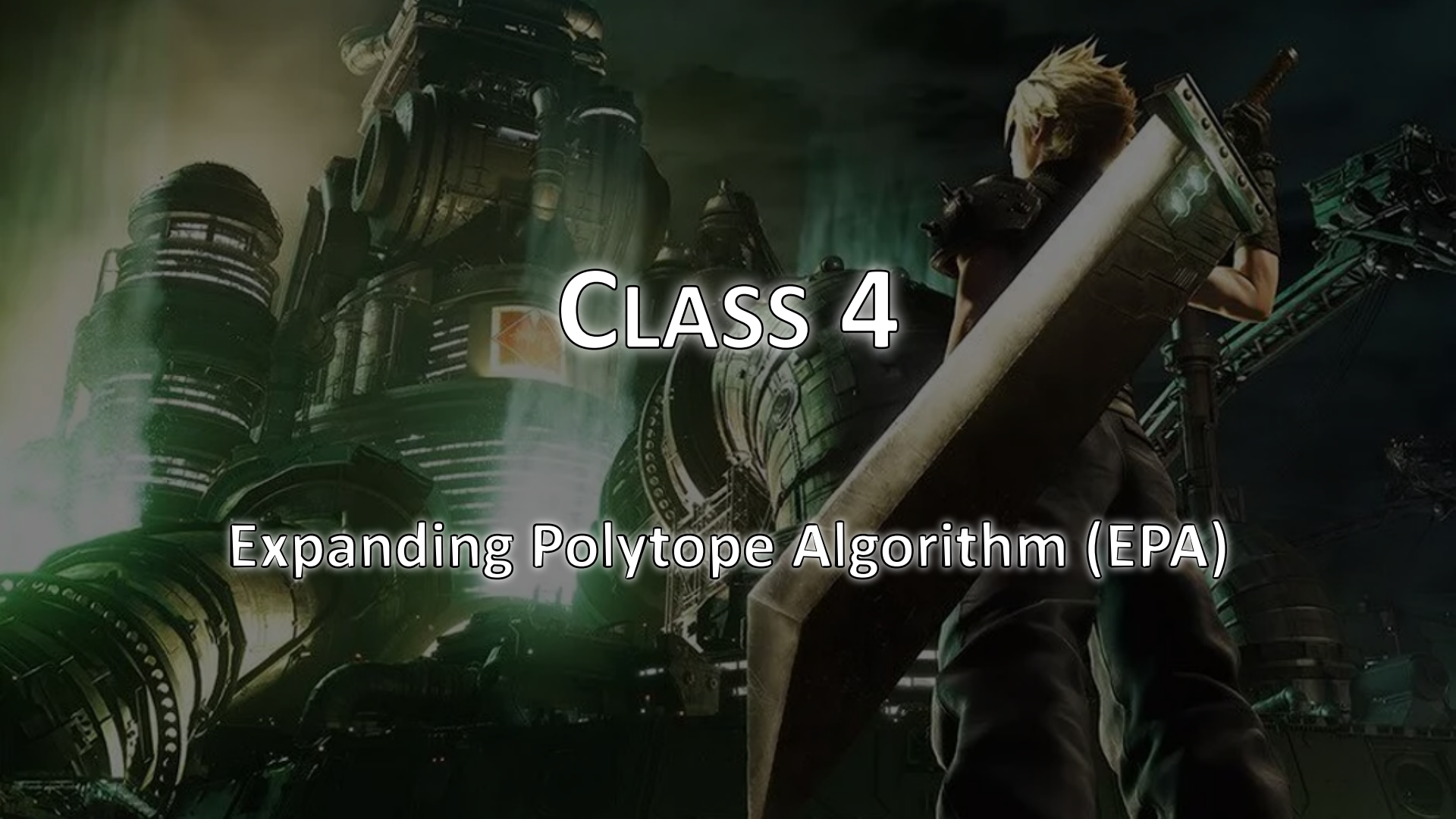
- The Minkowksi difference is in world space
- The support functions for our objects are in object space
- The direction *D* is in world space
- Then how do we calculate the Minkowski difference?

Evaluating the Minkowski Difference Support function

1. We have objects 1 and 2
2. Convert *D* into the object space of objects 1 and 2 to get D_1 *and* D_2
3. Evaluate the support function of object 1 using D_1 and evaluate the support function of object 2 using D_2 to get points p_1 and p_2
4. Convert p_1 and p_2 to world space to get P_1 and P_2 and then return the difference of P_1 and P_2

Are we done yet?

- So, we have everything we need to implement the GJK algorithm!
- But the GJK algorithm gives us a boolean indicating whether there was collision (and a simplex containing the origin if there was a collision)
- Don't we want a minimum translation vector (MTV)?
  - We need to know how to resolve the collision once we know it is occurring
- Introducing…

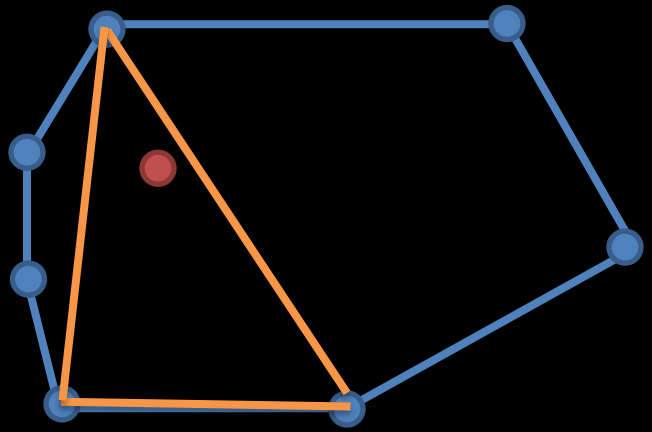

Class 4
Expanding Polytope Algorithm (EPA)

Expanding Polytope Algorithm

- If there was a collision, then we must resolve it (i.e. stop the objects from intersecting)
  - We get a simplex containing the origin from the GJK algorithm if there was a collision
- It turns out that the **minimum translation vector (MTV) is the vector connecting the origin to the point on the Minkowski difference closest to the origin**
  - In other words, if point *A* is the point on the Minkowski difference closest to the origin, then the MTV is *A*!
  - The goal of the Expanding Polytope Algorithm is to find the point on the Minkowski difference closest to the origin, so that we can use the MTV
- Note that it does not make sense to use the Expanding Polytope Algorithm if there was not a collision, because in that case there is no need for an MTV
- In fact, **the Expanding Polytope Algorithm (EPA) only works when we have a simplex of points in the Minkowski difference and this simplex contains the origin**

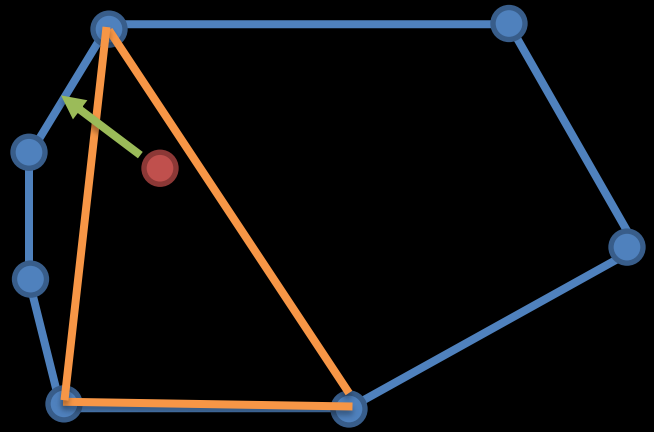
Expanding Polytope Algorithm

- Let the blue shape be the Minkowski difference, the orange shape be the simplex, and the red point be the origin



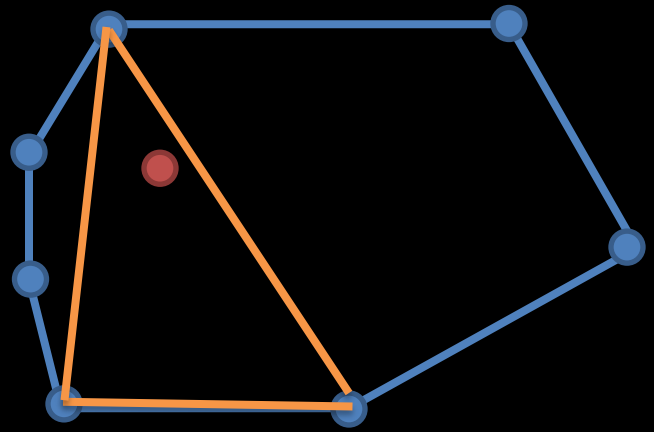
Expanding Polytope Algorithm

- We can see that the MTV is the green vector below



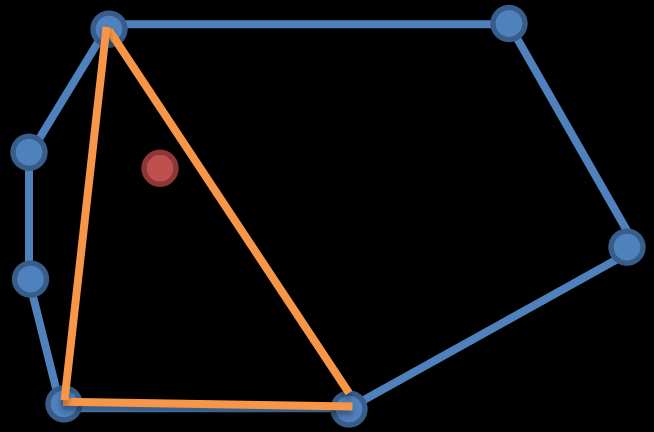
Expanding Polytope Algorithm

- We can see that the MTV is the green vector below



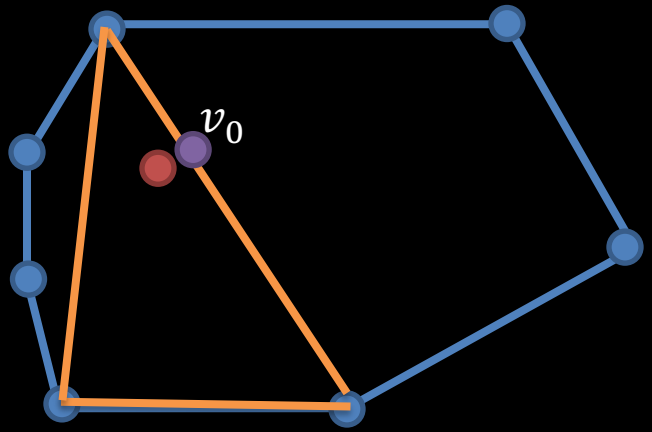
Expanding Polytope Algorithm

- EPA iteratively expands the simplex, adding more points on the boundary of the Minkowski difference to find the MTV



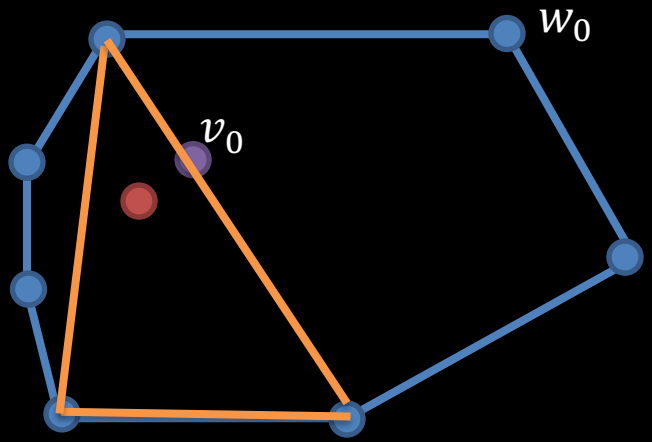
Expanding Polytope Algorithm

- To find the next point on the boundary of the Minkowski difference, we find the point on the boundary of the simplex (which we will now call a polytope) closest to the origin (the purple point)
- At iteration i, call this point v_i

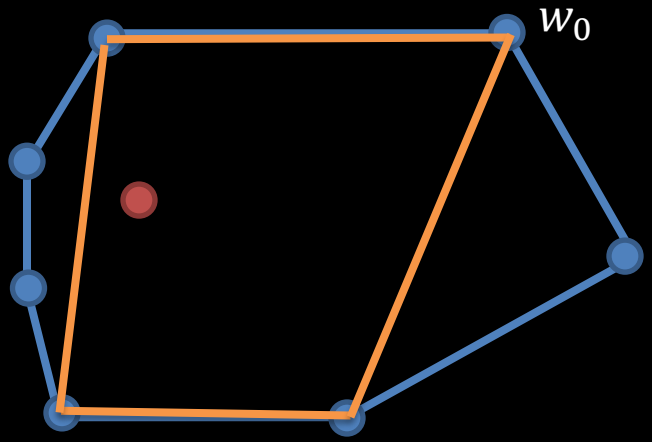


Expanding Polytope Algorithm

- We plug v_i into the support function of the Minkowski difference to get w_i
- We terminate the algorithm and return v_i when the projection of w_i onto v_i is equal to v_i (within some tolerance)
- When the projection of w_i onto v_i is equal to v_i, we know that the v_i is on the boundary of the Minkowski difference
- In this iteration, we see that this termination condition is not met, so we add w_i to the polytope by splitting the edge/face that v_0 belonged to

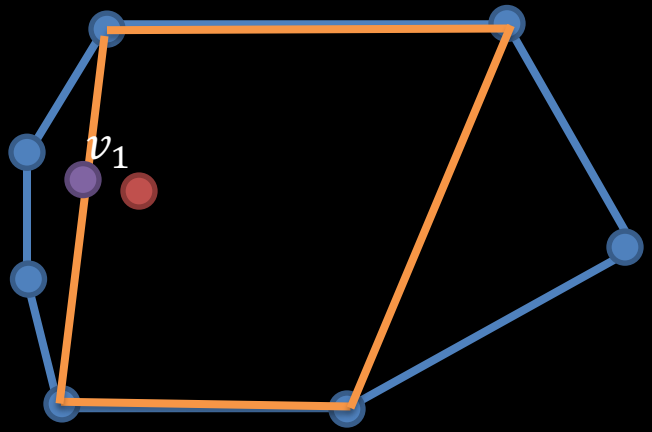


Expanding Polytope Algorithm



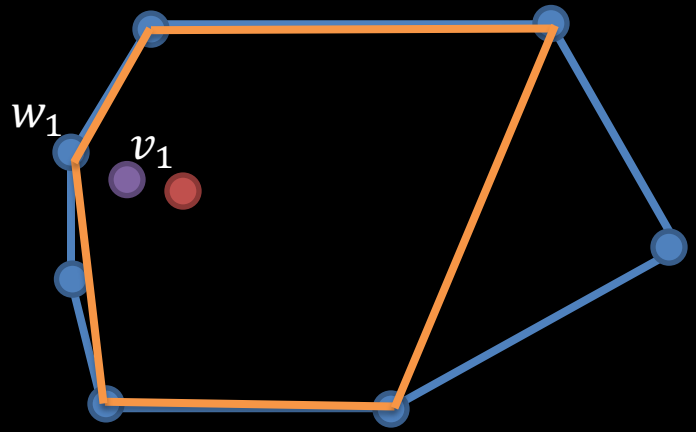
Expanding Polytope Algorithm

- Next, we carry out another iteration
- We find the point on the boundary of the polytope closest to the origin (v_1)



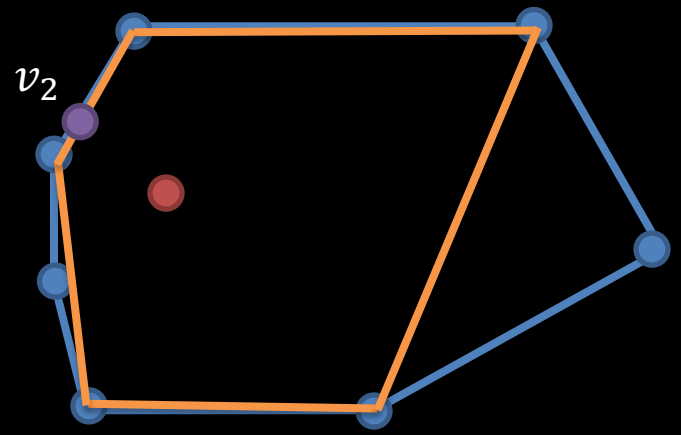
Expanding Polytope Algorithm

- Then, we plug v_1 into the support function of the Minkowski difference to get w_1
- The projection of w_1 onto v_1 is not equal to v_1, so we add w_1 to the polytope by splitting the edge v_1 belonged to



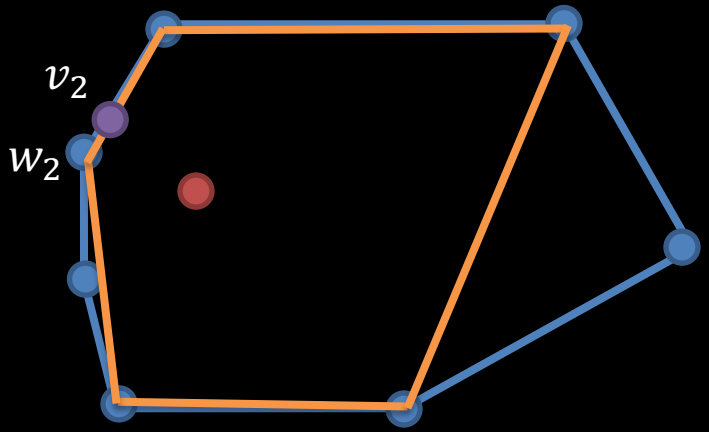
Expanding Polytope Algorithm

- We find the point on the polytope closest to the origin (v_2)



Expanding Polytope Algorithm

- Then we plug v_2 into the support function of the Minkowski difference to get w_2
- The projection of of w_2 onto v_2 is equal to v_2, so we return v_2, which is the MTV!



EPA Pseudocode

```
EPA(simplex):
  polytope = simplex
  while (true)
    face = getClosestFaceToOrigin(polytope)
    v = projectionOfOriginOnFace(face)
    w = minkowskiSupport(v)
    if projection(w,v) == v:
      return v
    else
      simplex.add(w)
```

Extra Details

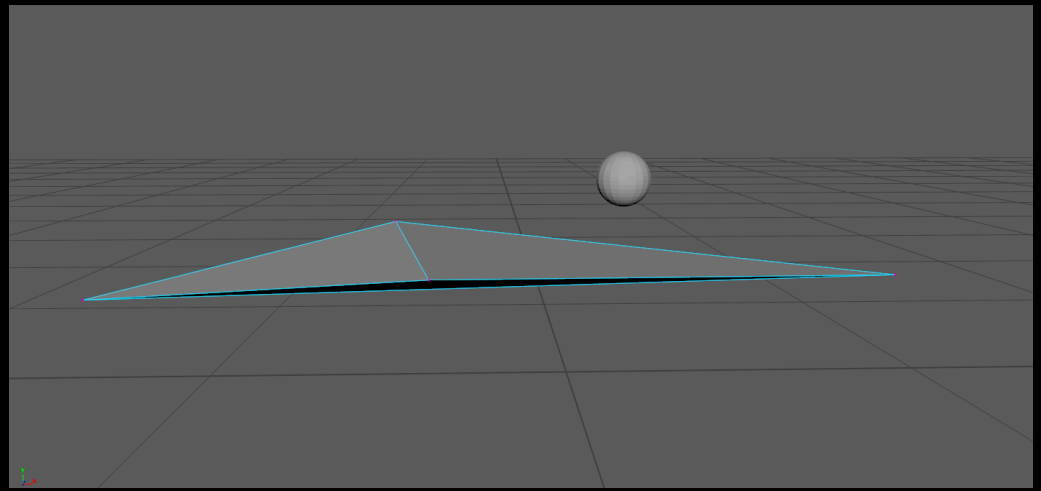
- We have looked at an EPA example in 2D
- There are a few details that we need to consider when we transition to 3D

Representing the Polytope

- We have looked at an EPA example in 2D
- There are a few details that we need to consider when we transition to 3D
- First, the polytope in 3D is defined as a set of triangles forming a convex 3D shape
  - We can represent the polytope like a mesh, by having a list of vec3s defining the vertex positions, and a list of triplets of integers representing the faces

Expanding the Polytope in 3D

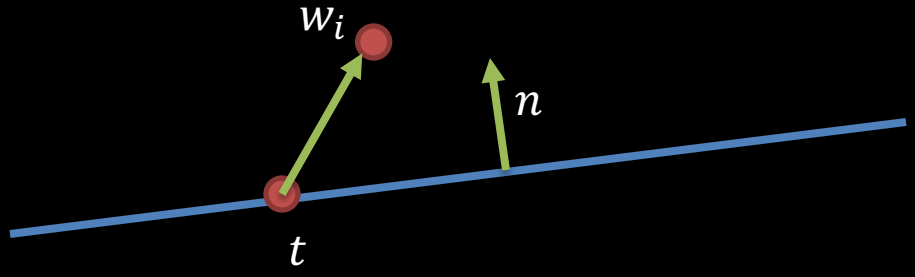
- If we need to expand the polytope, we can't just split the face containing v_i into 3 faces that contain w_i
- If we do this, we will get a non-convex shape, as shown in the video below (the sphere represents w_i)



Expanding the Polytope in 3D

- To handle this problem, we need to make sure every face of the polytope that "sees" w_i is changed so that face has w_i as a vertex
- A face with normal $\boldsymbol{n}$ and a vertex t "sees" w_i if $(w_i - t) \cdot \boldsymbol{n} > 0$

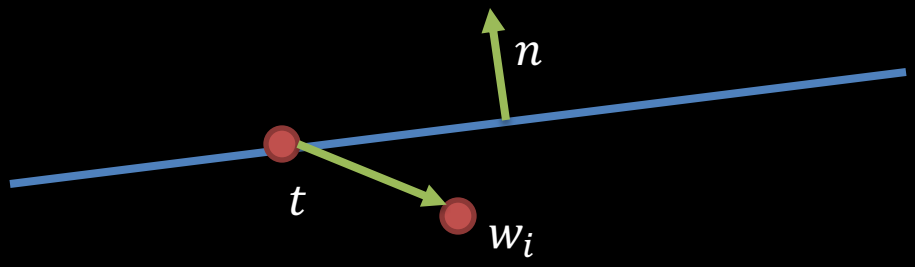
Here, the blue face "sees" w_i



Expanding the Polytope in 3D

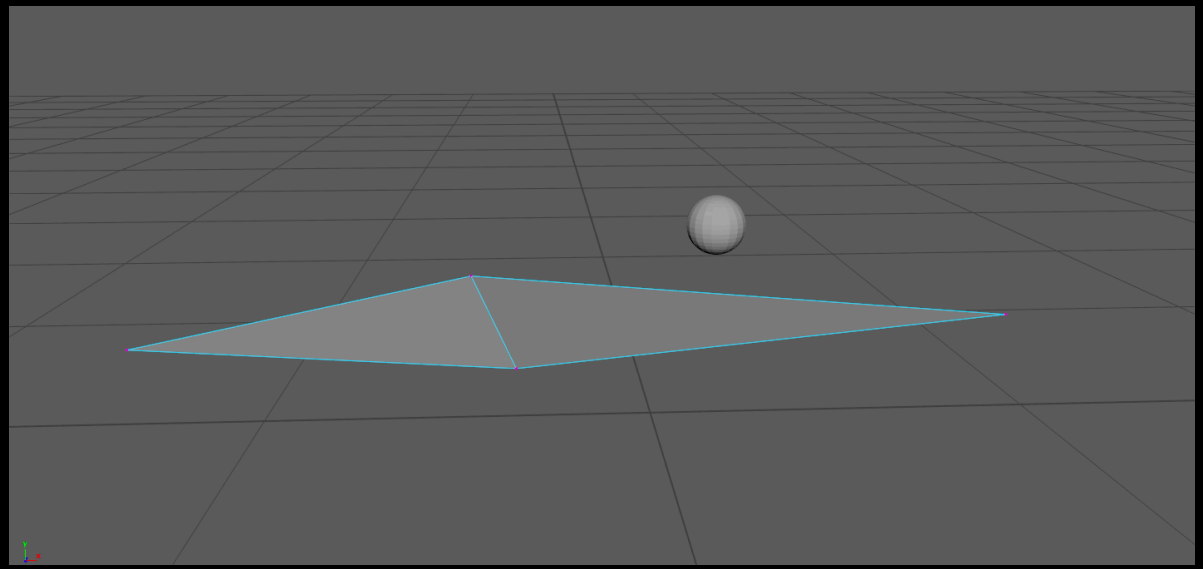
- To handle this problem, we need to make sure every face of the polytope that "sees" w_i is changed so that face has w_i as a vertex
- A face with normal $\boldsymbol{n}$ and a vertex t "sees" w_i if $(w_i - t) \cdot \boldsymbol{n} > 0$

Here, the blue face **does not** "see" w_i



Expanding the Polytope in 3D

- The video below provides a demonstration of this process



Expanding the Polytope in Pseudocode

```
edges = []
for face in polytope:
  if face.sees(w):
    polytope.remove(face)
    for edge in face:
      if edge in edges:
        edges.remove(edge)
      else:
        edges.append(edge)
for edge in edges:
  polytope.addTriangle(Triangle(edge.start, edge.end, w))
```

Pseudocode from GJK + Expanding Polytope Algorithm – Implementation and Visualization by Andrew Smith

Expanding the Polytope Pseudocode

- The pseudocode says that any face that does not see w_i will be in the new polytope
- Any face that **does** see w_i will be removed
- For each edge of a face that sees w_i but is not shared between two faces that see w_i create a face using that edge and w_i
- This process results in a convex polytope

Expanding the Polytope Pseudocode

- You don't need to maintain a winding order for the triangles in your polytope
- **The origin must be inside the polytope, which means that the normal of a face of the polytope is always pointing away from the origin**
  - The normal $\boldsymbol{n}$ of a face with a vertex t must satisfy $t \cdot \boldsymbol{n} > 0$
  - If the normal you calculated using a cross product does not satisfy this requirement, just multiply the normal by -1!

Start with a 3-simplex

- You may have to deal with special cases if you start EPA with a simplex that is not a 3-simplex (tetrahedron)
- It is fine to force GJK to output a 3-simplex even if a 2-simplex containing the origin was found

Numerical Instability

- It is common for the origin to be very close to the edge of the Minkowski difference, which may cause numerical instability
- As a result, `length(projection(w,v) – v)` might not become as small as we want it to
- We can deal with this by keeping track of the smallest `length(projection(w,v) – v)` and returning the corresponding `v` if we run more than 10 iterations

Extra Details

- What if we want the point of contact between the two objects?
- We need the two points on each of the objects whose difference is the MTV!
- We know that the MTV is the point v on the polytope that we return from EPA
- Recall that if we have two objects A and B, then the Minkowski difference is $M_{A-B} = p_A - p_B$ for all points p_A in object A and all points p_B in object B

Extra Details

- Recall that the point v on the polytope that we return exists on a triangle whose vertices are points on the boundary of the Minkowski difference
  - Call these vertices $M_1, M_2, \text{and } M_3$
- Given two colliding objects *A* and *B*, we want two points A_* on object *A* and B_* on object *B* such that $A_* - B_* = v$
- We can define either A_* or B_* in world space as the point of collision
- Call these vertices $M_1, M_2, \text{and } M_3$
  - Using barycentric coordinates, we can say that $v = c_1M_1 + c_2M_2 + c_3M_3$, where $c_1 + c_2 + c_3 = 1$
  - Each $M_i = A_i - B_i$ where A_i is a point on object *A* and B_i is a point on object *B*
  - This means that we can say $A_* = c_1A_1 + c_2A_2 + c_3A_3$ and $B_* = c_1B_1 + c_2B_2 + c_3B_3$

Extra Details

- The following code calculates the barycentric coordinates of point *p* for a triangle with vertices *a*, *b, and c*

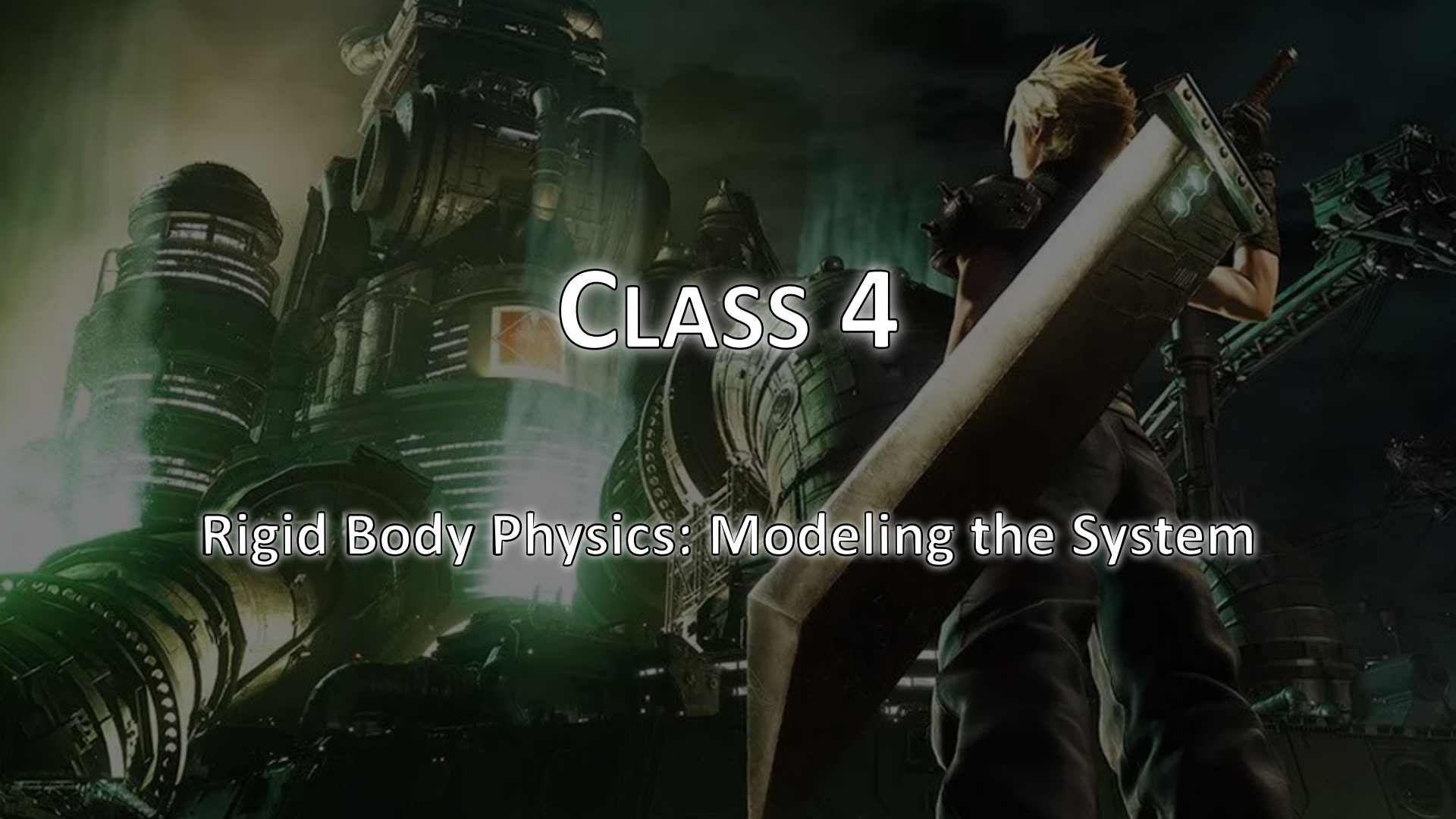
```
glm::vec3 Barycentric(glm::vec3 p, glm::vec3 a, glm::vec3 b, glm::vec3 c)
{
    glm::vec3 v0 = b - a, v1 = c - a, v2 = p - a;
    float d00 = glm::dot(v0, v0);
    float d01 = glm::dot(v0, v1);
    float d11 = glm::dot(v1, v1);
    float d20 = glm::dot(v2, v0);
    float d21 = glm::dot(v2, v1);
    float denom = d00 * d11 - d01 * d01;
    float v = (d11 * d20 - d01 * d21) / denom;
    float w = (d00 * d21 - d01 * d20) / denom;
    return glm::vec3(1.0f - v - w, v, w);

}
```

Extra Details

- We can test collisions between a convex shape and a triangle!
  - The support function of a triangle is just the vertex of the triangle farthest in the direction of the input vector!

Hooray!

- Now we have everything we need to implement basic rigid body physics for arbitrary convex shapes!



Class 4

Rigid Body Physics: Modeling the System

Rigid Body Physics

- When we simulate physics on a computer, we represent the system we are simulating with a **state vector**
- Given this state vector and the forces acting on the system, we can calculate the derivative of the state vector and use Euler's method to propagate the system through time (i.e. next_state = old_state + derivative * dt)
- Keep in mind that there are other ways to implement the basics of rigid body physics
  - This presentation is demonstrating a method based on David Baraff's *An Introduction to Physically Based Modeling*

State Vector

- Here is our state vector for a single rigid body
- x(t) is the world space position of the object's center of gravity
- R(t) is the 3x3 rotation matrix that transforms the rigid body from object space to world space (we will call this the orientation matrix)
- P(t) is the object's linear momentum (which is a vector)
- L(t) is the object's angular momentum (which is a vector)

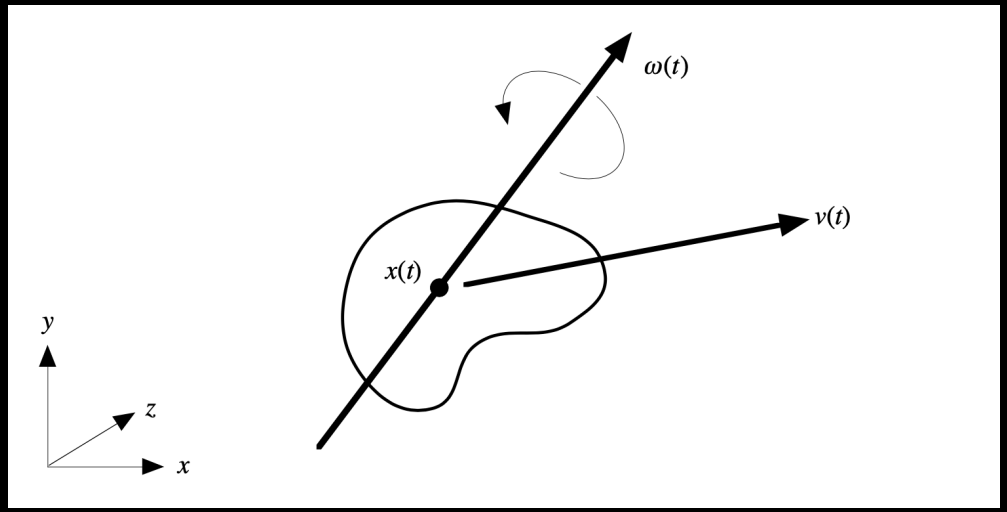
$$\mathbf{Y}(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}.$$

Rotations

- We need to discuss the physics of rotations a bit

Angular Velocity

- Angular velocity ω is a vector whose direction describes the axis around which an object rotates, and whose magnitude describes how fast the object is rotating



Angular Velocity

- We can use the angular velocity of an object to calculate the derivative the of the orientation matrix *R*

Given the vector a, let us define a^* to be the matrix

$$\begin{pmatrix} 0 & -a_z & a_y \\ a_z & 0 & -a_x \\ -a_y & a_x & 0 \end{pmatrix}.$$

$$\dot{R}(t) = \omega(t)^* R(t).$$

Torque

- If a force is exerted on an object that causes the rotation of that object to change, then we say that a **torque** is exerted on the object
- Consider the blue bar secured to the red hinge below

Torque

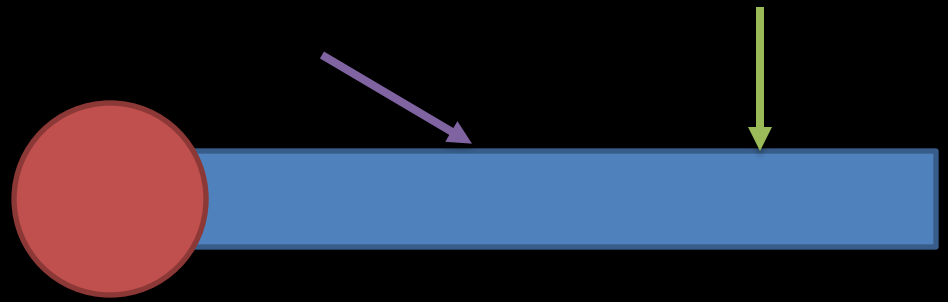
- If a force is exerted on an object that causes the rotation of that object to change, then we say that a **torque** is exerted on the object
- Consider the blue bar secured to the red hinge below

This green force exerts a torque on the bar

Torque

- If a force is exerted on an object that causes the rotation of that object to change, then we say that a **torque** is exerted on the object
- Consider the blue bar secured to the red hinge below

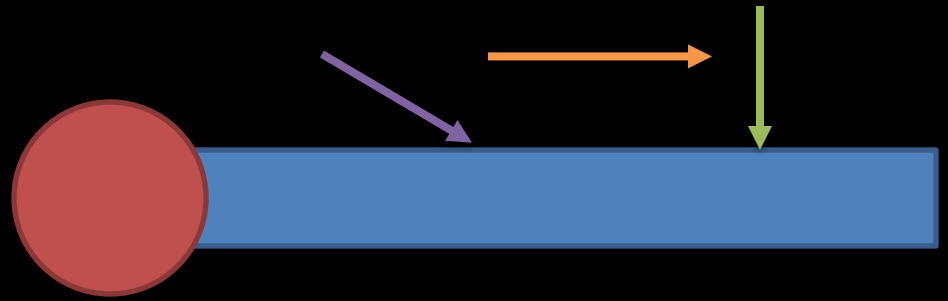
This purple force exerts a smaller torque on the bar, even though it is equal in magnitude to the green force



Torque

- If a force is exerted on an object that causes the rotation of that object to change, then we say that a **torque** is exerted on the object
- Consider the blue bar secured to the red hinge below

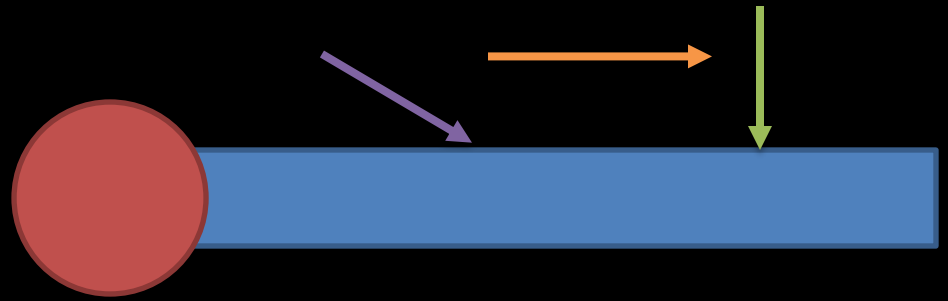
This orange force exerts no torque on the bar



Torque

- If a force is exerted on an object that causes the rotation of that object to change, then we say that a **torque** is exerted on the object
- Consider the blue bar secured to the red hinge below

This orange force exerts no torque on the bar



Angular Momentum

- Angular momentum is harder to understand than linear momentum
- You can think about it as a vector quantity describing how an object is rotating, and how difficult it is to stop the object from rotating
- Torque is the derivative of angular momentum

State Vector Derivative

- Here is our state vector for a single rigid body
- v(t) is the linear velocity
- $\omega(t) * R(t)$ is the derivative of the orientation matrix, as mentioned before
- F(t) is the force acting on the object
- $\tau(t)$ is the torque acting on the object

$$\frac{d}{dt}\mathbf{Y}(t) = \frac{d}{dt}\begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix} = \begin{pmatrix} v(t) \\ \omega(t)^* R(t) \\ F(t) \\ \tau(t) \end{pmatrix}.$$

State Vector Derivative

- We will pretend that the force and torque are given for now (you will probably only need to set the force to be gravity, and leave the torque alone)
- Here are the formulas for calculating the angular momentum and the linear velocity:

$$v(t) = \frac{P(t)}{M}$$

$$I(t) = R(t) I_{body} R(t)^T$$

$$\omega(t) = I(t)^{-1} L(t)$$

- Wait, what are $I(t)$ and I_{body}?

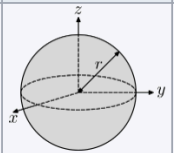
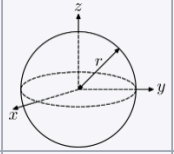
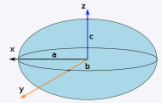
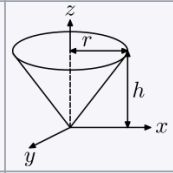
The Inertia Tensor

- I and I_{body} are the inertia tensors in world space and object space, respectively
- The inertia tensor is a 3x3 matrix
- The inertia tensor is a generalization of the "moment of inertia"
- I relates the object's angular velocity to its angular momentum, as shown below

$$\omega(t) = I(t)^{-1}L(t).$$

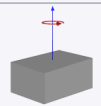
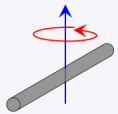
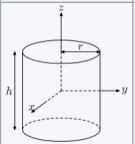
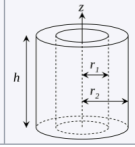
The Inertia Tensor

- Here are inertia tensors in object space for different shapes

Description	Figure	Moment of inertia tensor
Solid sphere of radius r and mass m		$I = \begin{bmatrix} \frac{2}{5}mr^2 & 0 & 0 \\ 0 & \frac{2}{5}mr^2 & 0 \\ 0 & 0 & \frac{2}{5}mr^2 \end{bmatrix}$
Hollow sphere of radius r and mass m		$I = \begin{bmatrix} \frac{2}{3}mr^2 & 0 & 0 \\ 0 & \frac{2}{3}mr^2 & 0 \\ 0 & 0 & \frac{2}{3}mr^2 \end{bmatrix}$
Solid ellipsoid of semi-axes a, b, c and mass m		$I = \begin{bmatrix} \frac{1}{5}m(b^2+c^2) & 0 & 0 \\ 0 & \frac{1}{5}m(a^2+c^2) & 0 \\ 0 & 0 & \frac{1}{5}m(a^2+b^2) \end{bmatrix}$
Right circular cone with radius r, height h and mass m, about the apex		$I = \begin{bmatrix} \frac{3}{5}mh^2 + \frac{3}{20}mr^2 & 0 & 0 \\ 0 & \frac{3}{5}mh^2 + \frac{3}{20}mr^2 & 0 \\ 0 & 0 & \frac{3}{10}mr^2 \end{bmatrix}$

The Inertia Tensor

- Here are inertia tensors in object space for different shapes

Solid cuboid of width w, height h, depth d, and mass m		$I = \begin{bmatrix} \frac{1}{12}m(h^2+d^2) & 0 & 0 \\ 0 & \frac{1}{12}m(w^2+d^2) & 0 \\ 0 & 0 & \frac{1}{12}m(w^2+h^2) \end{bmatrix}$
Slender rod along y-axis of length l and mass m about end		$I = \begin{bmatrix} \frac{1}{3}ml^2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{3}ml^2 \end{bmatrix}$
Slender rod along y-axis of length l and mass m about center		$I = \begin{bmatrix} \frac{1}{12}ml^2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{12}ml^2 \end{bmatrix}$
Solid cylinder of radius r, height h and mass m		$I = \begin{bmatrix} \frac{1}{12}m(3r^2+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3r^2+h^2) & 0 \\ 0 & 0 & \frac{1}{2}mr^2 \end{bmatrix}$
Thick-walled cylindrical tube with open ends, of inner radius r_1, outer radius r_2, length h and mass m		$I = \begin{bmatrix} \frac{1}{12}m(3(r_2^2+r_1^2)+h^2) & 0 & 0 \\ 0 & \frac{1}{12}m(3(r_2^2+r_1^2)+h^2) & 0 \\ 0 & 0 & \frac{1}{2}m(r_2^2+r_1^2) \end{bmatrix}$

The Inertia Tensor

- You can use these inertia tensors in object space along with your object's orientation matrix to calculate the inertia tensor in world space

$$I(t) = R(t) I_{body} R(t)^T$$

Euler's Method

- Now we know how to calculate the derivative of the state vector!
- We can use Euler's method to propagate the system through time
- next_state = old_state + derivative * dt

Euler's Method

- Just one problem…
- We need the orientation matrix to be a rotation matrix (i.e. its columns are unit vectors that are orthogonal to each other)
- If we add $\omega(t) * R(t) \cdot dt$ to the orientation matrix, we will not end up with a rotation matrix!
- The easiest thing to do is to use the Gram-Schmidt process
  - This will turn the columns of $R(\mathrm{t}) + \ \omega(t) * R(t) \cdot dt$ into a "close" orthonormal basis
  - There are better solutions to this problem

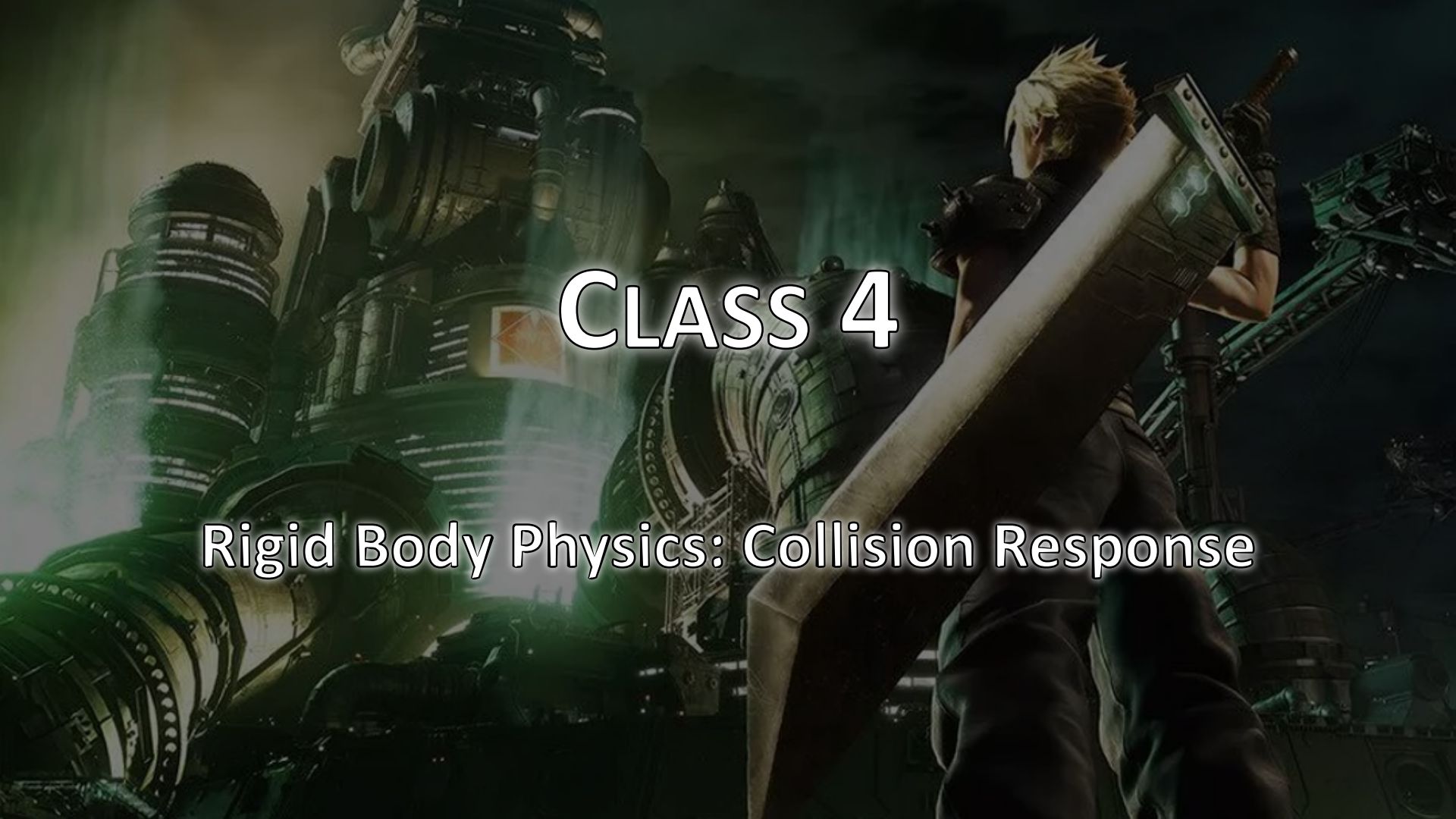
Gram-Schmidt Process

- This code takes in a matrix (in column-major order) and returns a new matrix whose columns are the result of applying the Gram-Schmidt process to the columns of the original matrix

```
glm::mat3 RigidBodyComponent::gramSchmidt(const glm::mat3 m) {
    glm::vec3 v0 = m[0];
    glm::vec3 v1 = m[1];
    glm::vec3 v2 = m[2];

    glm::vec3 w0 = glm::normalize(v0);
    glm::vec3 w1 = glm::normalize(v1 - glm::dot(v0, v1) / glm::dot(v0, v0) * v0);
    glm::vec3 w2 = v2 - (glm::dot(v0, v2) / glm::dot(v0, v0) * v0) - (glm::dot(v1, v2) / glm::dot(v1, v1) * v1);

    glm::mat3 ret;
    ret[0] = w0;
    ret[1] = w1;
    ret[2] = w2;

    return ret;
}
```

What Happened to Collisions?

- We still haven't discussed how we use the MTV and collision point to deal with collisions!



Class 4

Rigid Body Physics: Collision Response

Impulse

- Impulse is a change in momentum due to a force
- It is equal to the force times the time interval over which it acts
- When a collision happens, we will apply an impulse to the colliding objects

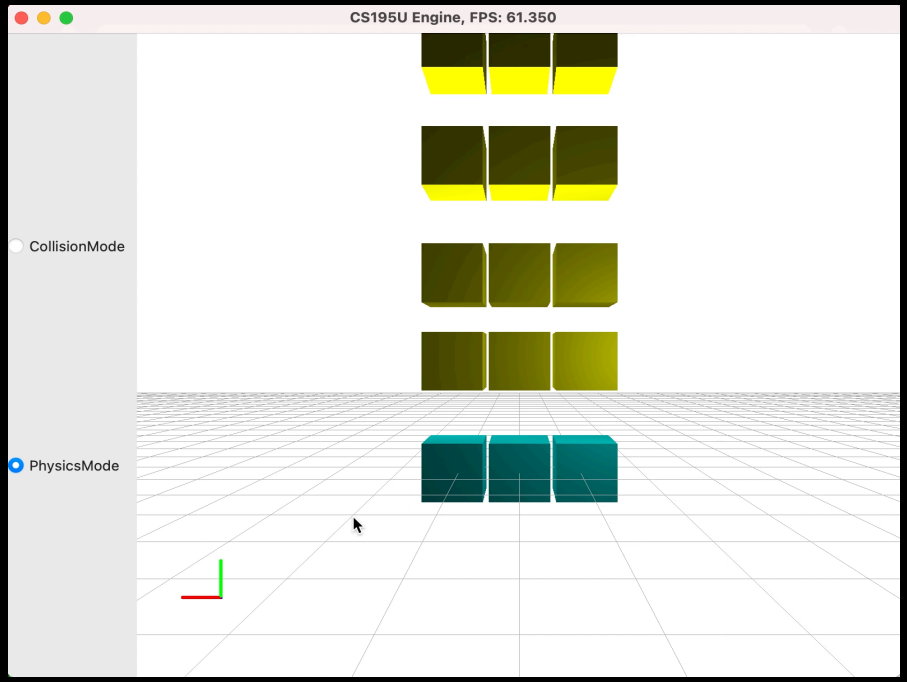
$$\Delta p = F\Delta t$$

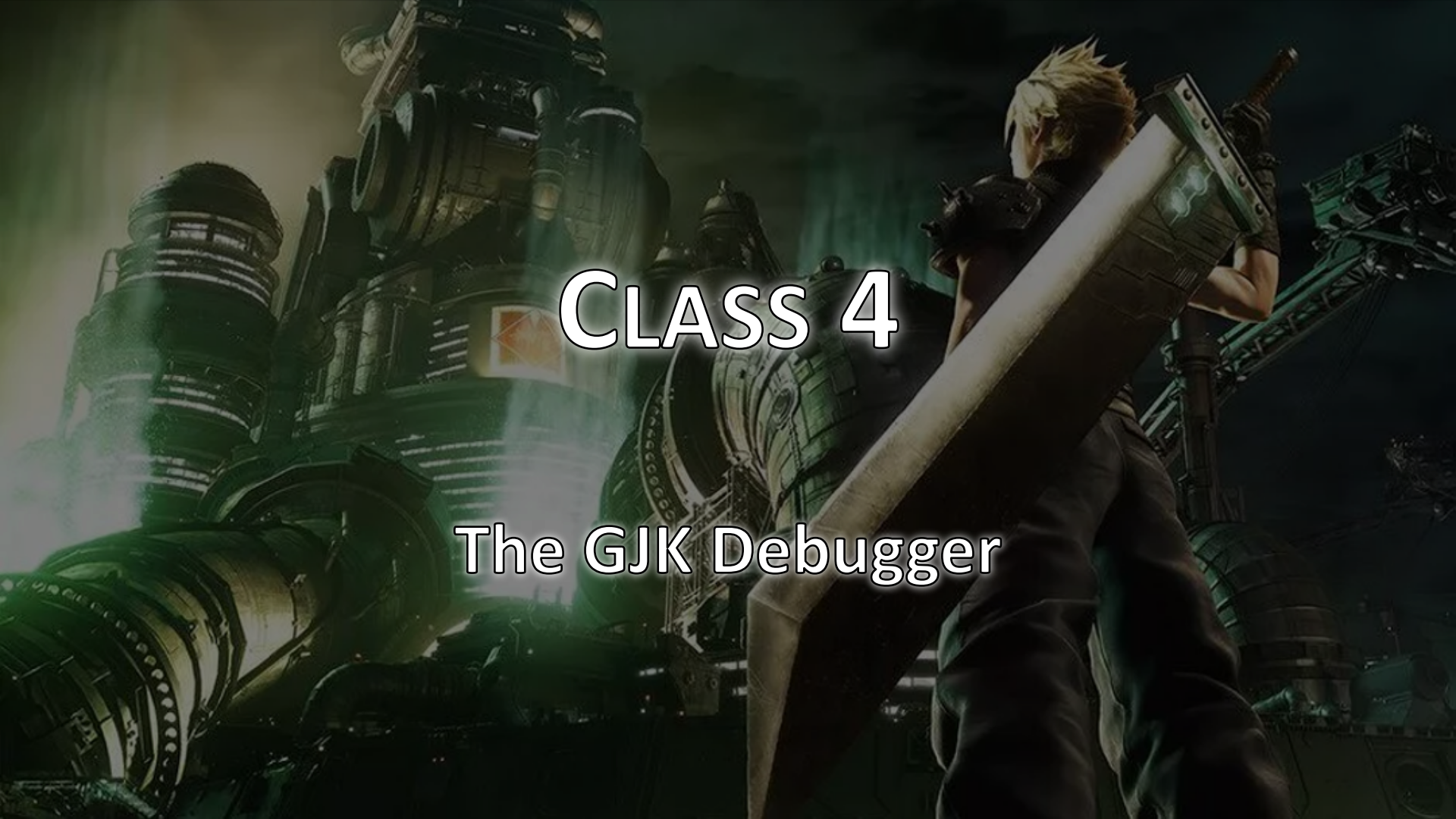
Impulse

- Impulse is a change in momentum due to a force
- It is equal to the force times the time interval over which it acts
- When a collision happens, we will apply an impulse to the colliding objects using the mtv, the points of collision of the two objects, and those two points' velocities
- We can apply linear and rotational impulse
- Once we have applied the impulse, we use Euler's method again
- For more info, check out **An Introduction to Physically Based Modeling: *Rigid Body Simulation II—Nonpenetration Constraints by David Baraff***

Finally!

- We get collisions and rotational physics! Wow!
- Notice that there is no friction here

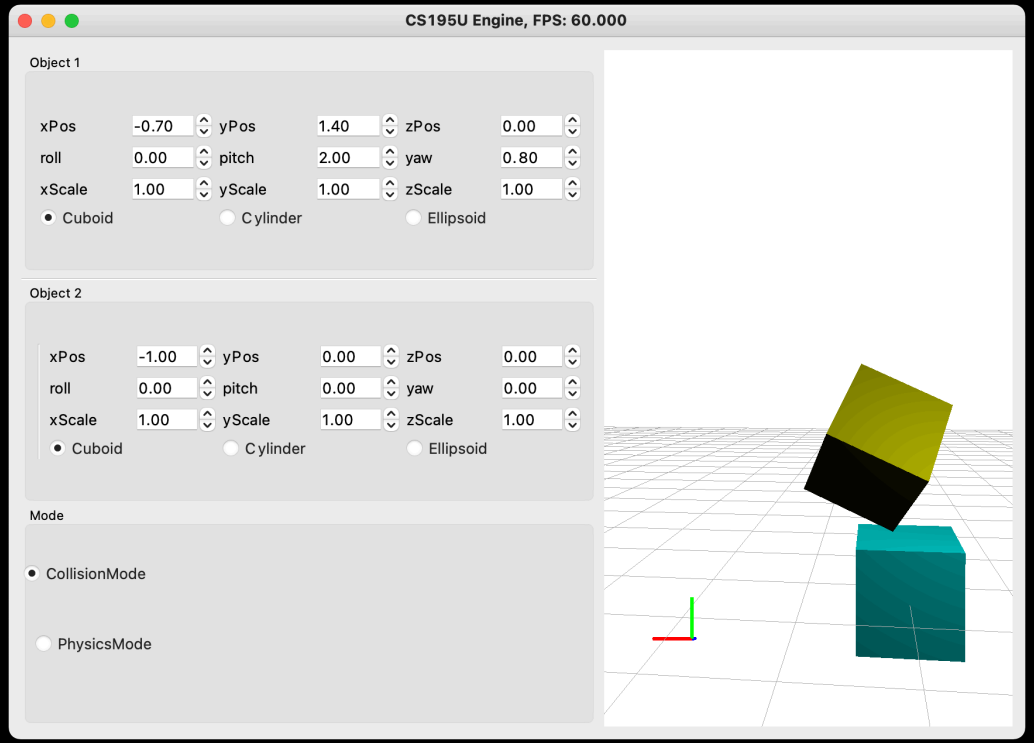


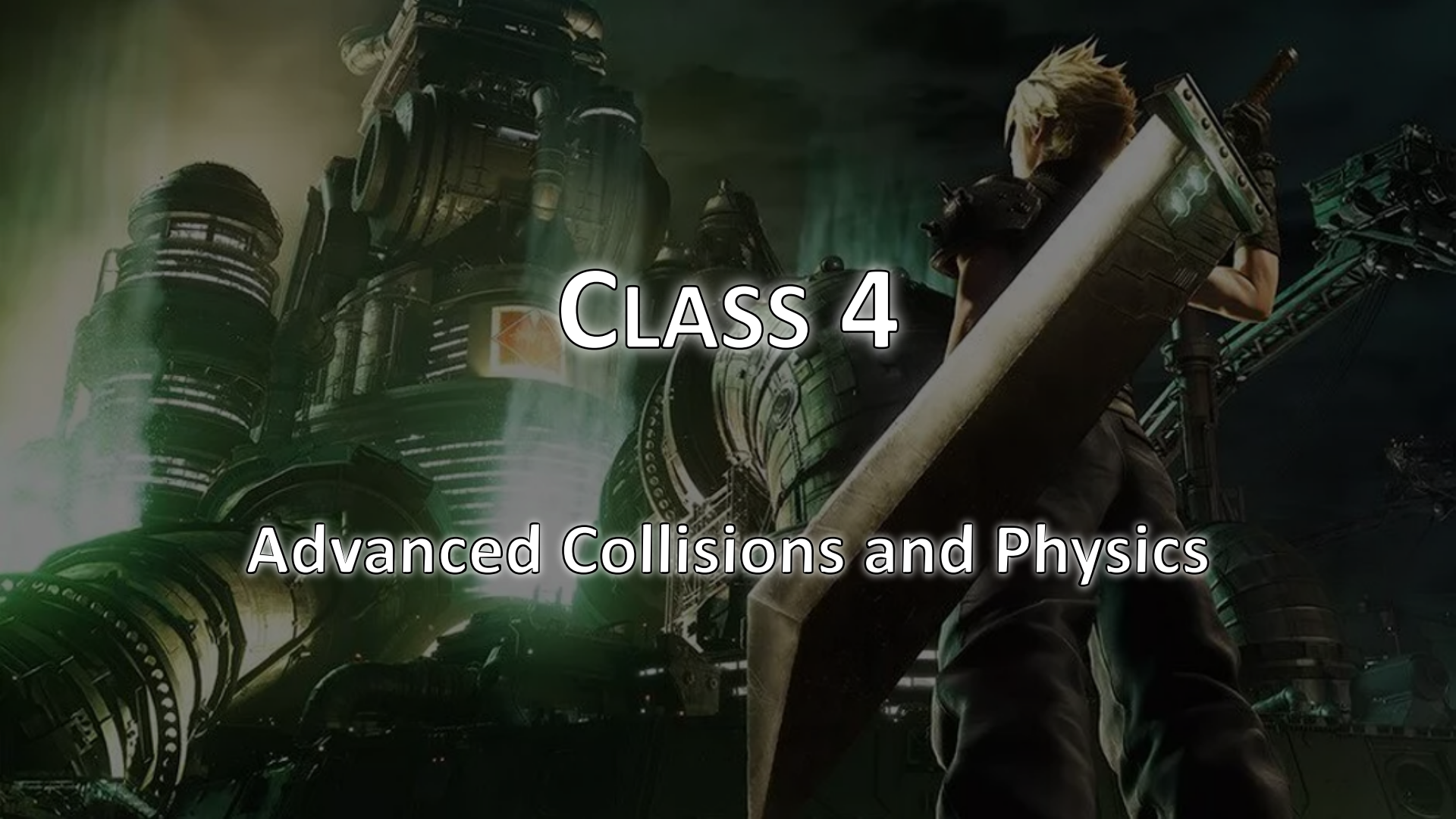

Class 4
The GJK Debugger

The GJK Debugger

- We have a debugging tool to help you implement the GJK algorithm!

The GJK Debugger




Class 4
Advanced Collisions and Physics