

# Implementing Quickhull

Dirk Gregorius – Valve Software

Good afternoon! My name is Dirk and I am a software engineer at Valve.

Some of you might have noticed that I changed the topic of my talk to:  
“Implementing Quickhull”

When I was rehearsing the talk at Valve one question was: “Why a presentation about Quickhull? Isn’t that a well understood algorithm?”

From my experience this is only partially true. You find some presentations about the basic theory of Quickhull and some complexity analysis, but there is hardly anything that deals with **implementing** this algorithm. Especially not in a production environment where you have to deal with all sorts of ill-defined input geometry! So this talk will cover the basic theory of Quickhull of course, but we will also spend a good amount of time looking into problems you can run into when **implementing** Quickhull and how to address those.

# Outline

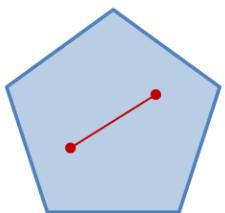
---

- **Introduction**
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

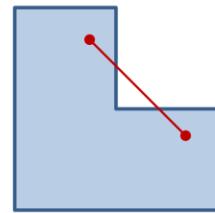
Before we start I like to quickly outline the talk:

- After a short introduction I will first start with Quickhull in 2D
- Then we continue with geometrical invariants which we need to maintain while constructing the hull to avoid numerical problems
- After the 2D introduction we will directly dive into the 3D version of Quickhull
- We then investigate geometrical and topological invariants while constructing the hull in 3D and close with some implementation details

# Convexity



Convex



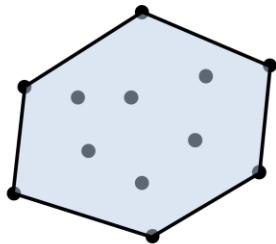
Concave

Before we start let's figure out what a convex hull is and look at an example:

- We call a shape convex, if for any two points that are \*inside\* the shape, the line between these two points is also inside the shape
- If you look at concave case you see that it is easy to find two points inside the shape where the line has to leave and then enter the shape again.

# Convex Hulls

---



A 2D set of points and its convex hull

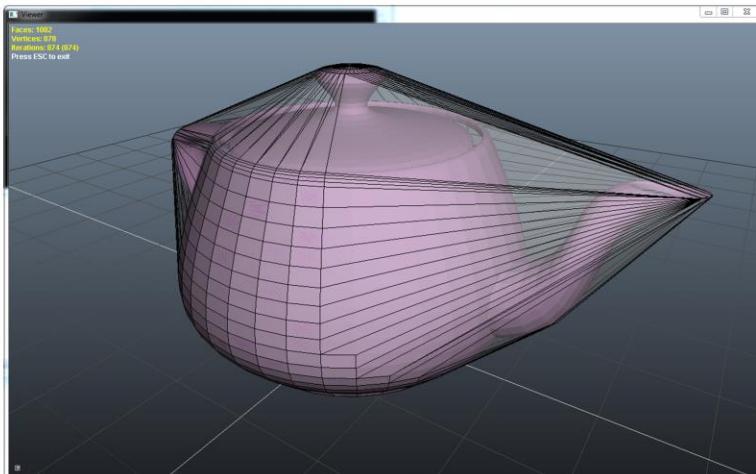
What is a convex hull?

Given a set of  $N$  input points we can now define what we are going to call a convex hull:

- Formally: A convex hull is the smallest convex set containing all input points
- Informally: If your points would be nails sticking in some piece of wood, the convex hull would be a rubber band wrapped around the outside nails.

**-> This means in 2D the hull is a convex polygon defined by vertices and edges!**

## Convex Hulls (2)



I also like to show a 3D convex hull of a well known object – Utah Teapot.

- In 2D we used the rubber band analogy to get some intuition for a convex hull
- In 3D you can think of shrink wrapping the object

**-> In 3D the hull is a convex polyhedron defined by vertices, edges and \*polygonal\* faces! Note that we are not going to restrict ourselves to triangles faces only!**

# Convex Hulls for Collision Detection

---

- Dynamic game objects are usually approximated by simple shapes for collision detection
- Convex hulls are a good candidate for this
- Collision detection for convex polyhedra is well defined and robust (e.g. GJK and SAT)

Why should we use convex hulls for collision detection in games?

- Dynamic game objects are usually approximated by simpler shapes for collision detection since using the render geometry would NOT be efficient
- Convex hulls are a good candidate since they can approximate even complex geometry quite well
- Also, collision detection for convex polyhedra is well defined and robust. Think of GJK and SAT which we discussed already here in earlier tutorials

# Convex Hulls in the Game



Before we start I like to show two videos to give you some idea how convex hulls are used in games:

- Show a movie of convex hulls in the game
- Show how the physics engine sees the game

Sergiy will show you how to implement awesome physics visualization right after this talk!

# Quickhull

---

- Published by C. Barber and D. Dobkin in 1995
- Iterative algorithm that adds one point at a time
- Most importantly it addresses issues with ill-defined input sets while computing the convex hull
- The output is then a set of ‘fat’ faces that encloses the exact convex hull

I hope the videos gave you an idea about the problem we are trying to solve here today.

When I started looking in convex hulls I quickly came across an algorithm called Quickhull:

- Quickhull was published by Barber and Dobkin in 1995
- It is essentially an iterative algorithm that adds individual points one point at a time to an intermediate hull.
- When implementing an algorithm to build convex hulls you have to deal with input geometry that pushes the limit of floating point precision.
- Quickhull uses so called fat planes and face merging to address these problems
- The output is then a set of ‘fat’ faces that encloses the exact convex hull

In the remainder of the talk will try to explain what this means in detail!

# Outline

---

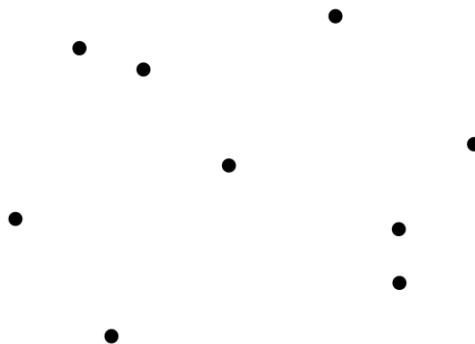
- Introduction
- **Introduce Quickhull in 2D**
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

I will start outlining the algorithm in 2D first

- I like to mention, that this is not the 'real' **2D** Quickhull algorithm (which actually exists). You should think of it as an introduction of the 3D version we will investigate later in the talk
- Personally I find it often helpful to think about things in 2D first, to get a good understanding of the problem and to familiarize myself with the basic ideas

# Quickhull 2D: Initial Hull

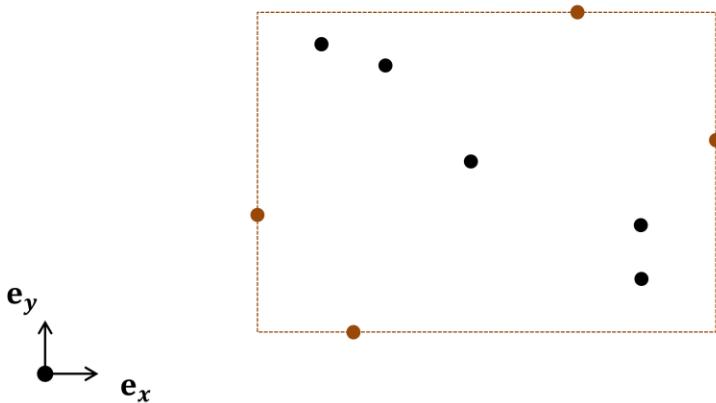
---



Assume we are given a set of points and we are asked to build the convex hull using the Quickhull algorithm:

- The first thing we need to do in Quickhull is to build an initial hull from where we can start adding points iteratively

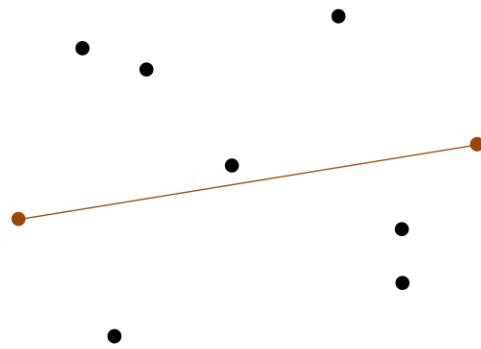
# Quickhull 2D: Initial Hull (2)



To find this initial hull we start by identifying the extreme points along each cardinal axis

- This simply means we find the points with the smallest and largest x and y values.

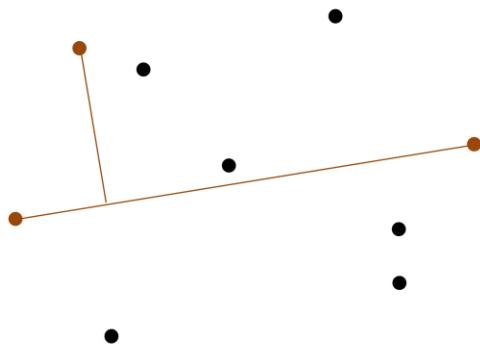
## Quickhull 2D: Initial Hull (3)



From these four points we choose the pair which is furthest apart

- In this example this would be the left- and right-most points

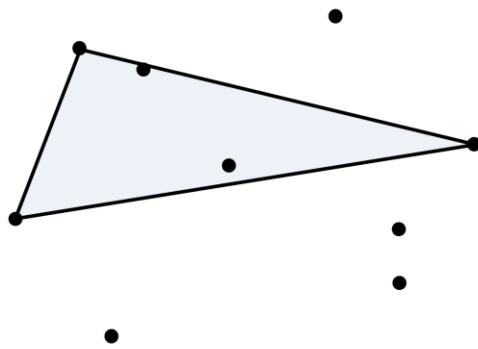
## Quickhull 2D: Initial Hull (4)



Finally we search for the furthest point from the line through these two extreme points

## Quickhull 2D: Initial Hull (5)

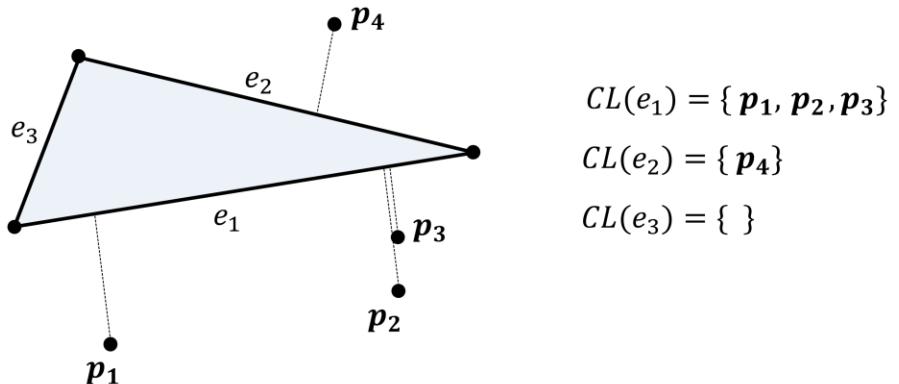
---



These three points build our initial hull

- In 2D the initial hull is simply a triangle

# Quickhull 2D: Partition Points



Before we start adding new points to the initial hull we have to do some book-keeping work:

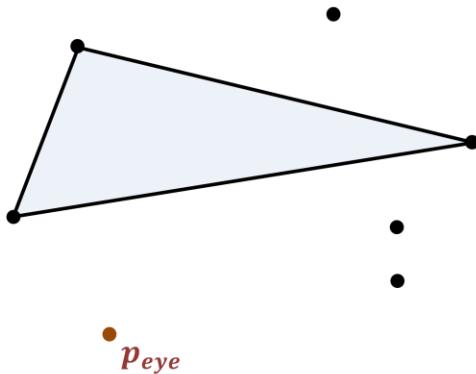
- The next step is to partition the remaining points and assign each point to its closest face
- We can also remove internal points since those cannot be on the final hull

What that means is that each face maintains a list of points which are outside the face plane. We call those ‘conflict lists’ since the points can “see” the face and therefore are potentially on the final hull. This is a clever way of managing the vertices since we don’t need to iterate all vertices when adding a new vertex to the hull. This makes Quickhull typically  $O(n \log n)$  in both 2 and 3 dimensions!

## NOTE:

Please don’t get confused here. Since I am presenting in 2D and 3D and some terminology overlaps I will use the terms ‘Edge’ and ‘Face’ interchangeable! This will usually help when we go to 3D later in the talk!

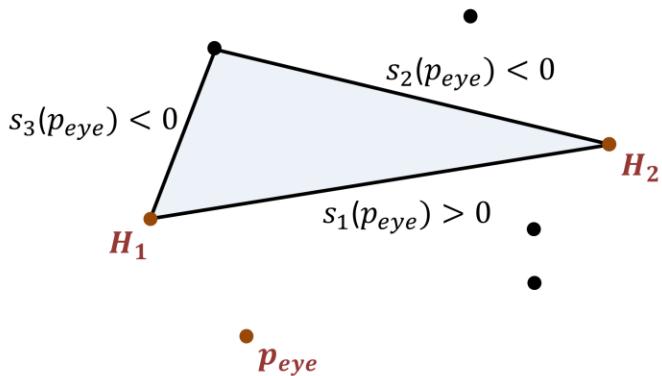
# Quickhull 2D: Find Next Vertex



The next step is to add a new point to our intermediate hull. We iterate our conflict lists and find the point  $p$  with the largest distance from the hull:

- Let's call this point the **eye** point
- Adding this new point requires several sub steps

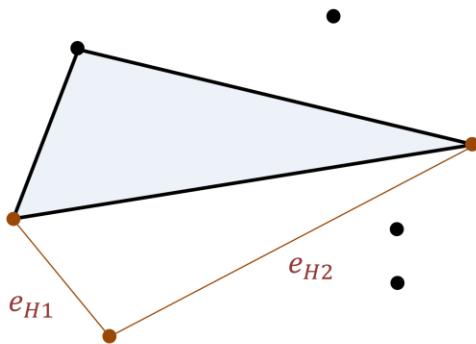
# Quickhull 2D: Find Horizon



First we need to identify all faces that are visible from the newly added point since these faces cannot be on the hull:

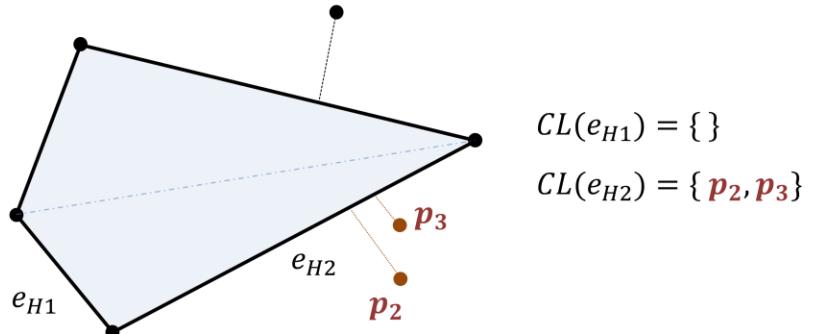
- A face is visible if the new point is in front of the face plane.
- We can use simple **plane tests** to classify the new point against each face!
- The next step is then to find the two vertices that connect a visible with a non-visible face.
- We call these two vertices the horizon
- The horizon is essentially the boundary between the visible and non-visible part of the current hull as seen from the new eye point

# Quickhull 2D: Add Point to Hull



Once we identified the two horizon vertices we then create two new faces for each horizon vertex to connect the new vertex to hull

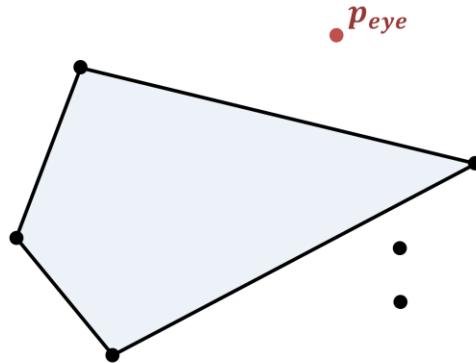
# Quickhull 2D: Partition Orphans



After building the new faces some old faces became obsolete

- Before we can delete these faces we need to handle their conflict lists since these conflict points can still be on the final hull
- We handle this by simply partitioning these orphaned vertices to the new faces

# Quickhull 2D: Add Point to Hull

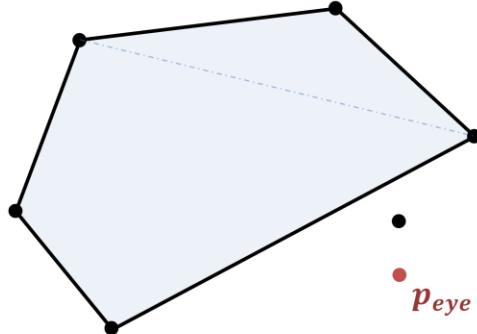


Finally we can now remove all old faces which were visible from the new point and therefore cannot be on the hull anymore

- This closes the iteration and we repeat those steps until all conflict list are empty
- In our example we continue and grab the next eye point and add it to hull as we just learned

# Quickhull 2D: Repeat (1)

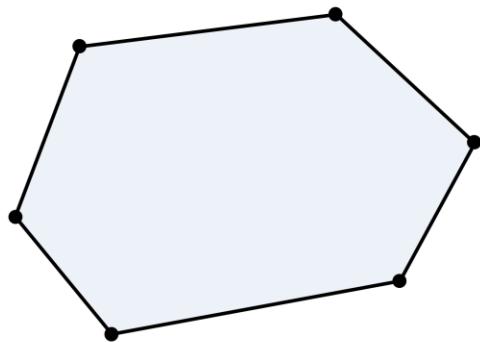
---



And we do it one more time to find our final hull...

## Quickhull 2D: Repeat (2)

---



When there are no more vertices (which means all conflict lists are empty) we are done!

# Quickhull 2D: Summary

- As you can see the basic Quickhull algorithm is pretty simple
- In 3D one major implementation difficulty arises from managing the lists of vertices, edges, faces and conflicts
- The other difficulty is dealing with numerical imprecision in our plane tests.

As you can see the basic ideas should be pretty easy to understand

- In 3D the major implementation difficulties actually arise from managing the lists of vertices, edges, faces and conflicts
- The other difficulty is dealing with numerical imprecision when classifying points using plane tests.

# Outline

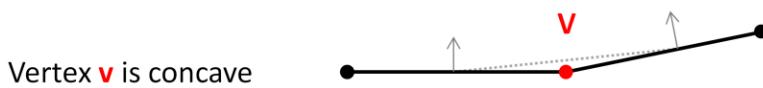
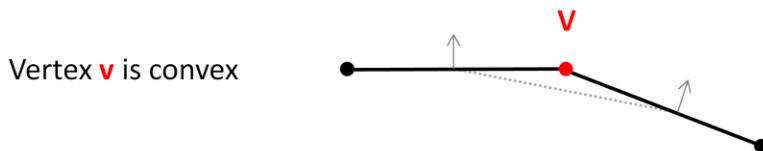
---

- Introduction
- Introduce Quickhull in 2D
- **Quickhull 2D Invariants**
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- Implementation

So far we pretended that our mathematical operations are exact

- Of course this is *\*not\** true in floating point arithmetic.
- Let's investigate how we can deal with those problems.

# Quickhull 2D: Invariants

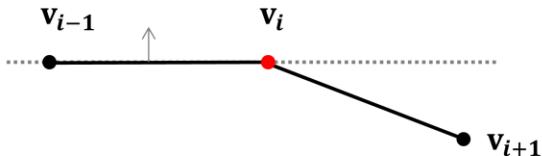


When building a convex hull we must maintain geometric invariants during construction:

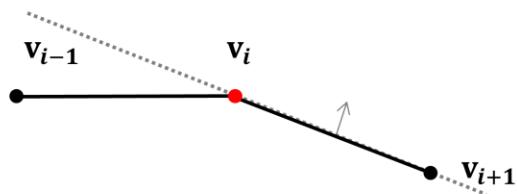
- In 2D we must guarantee that every vertex is convex
- This should be obvious since otherwise it would be simple to find a line between two points inside the hull that would leave and enter as shown in the slide at the beginning of the talk
- I tried to hint this with the dotted line between the **outside pointing** normals on the slide

# Quickhull 2D: Convexity Test

Left Face Test



Right Face Test

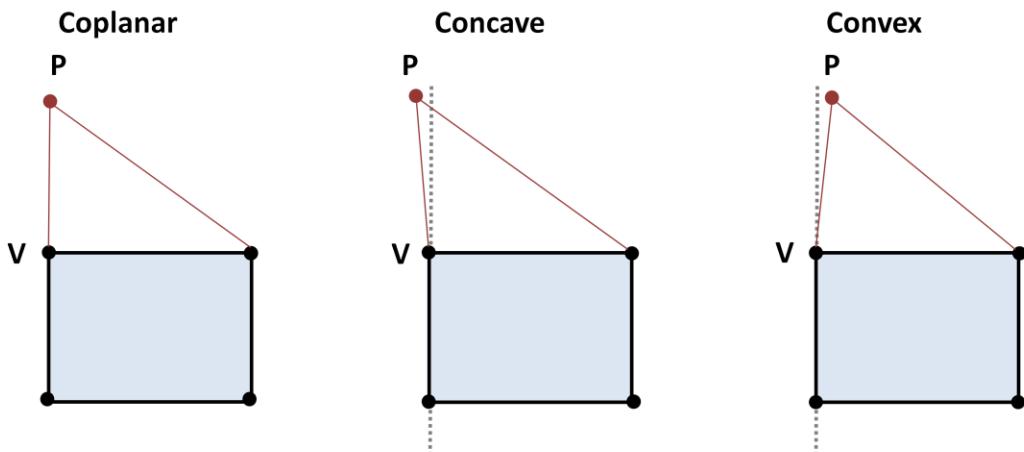


Next we need to define what a **convex** vertex is and how we can test a vertex for convexity:

For each vertex:

- First test if right vertex is below the left face
- Then test if left vertex if it is below the right face
- If both tests are true the vertex is **convex**, otherwise it must be **concave** or **coplanar**

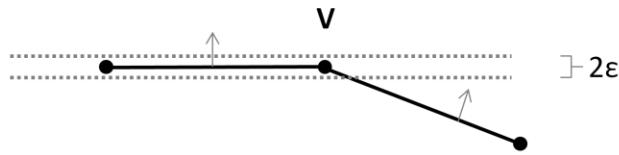
# Quickhull 2D: Numerical Robustness



Let's now look at an example where non-convex vertices might become an issue:

- Whenever we add a point which is collinear with an existing face things can become fuzzy
- A small variation of point **P** will define whether the vertex **V** will remain on the hull or not
- Ideally we would like to have more stability such that for very *small* variations within some tolerance we would get the same result
- Note that the point **P** is not actually moving, but can end up on either side of the plane *\*just\** due to numerical imprecision

# Quickhull 2D: Fat Planes



$$P := \mathbf{n} \cdot \mathbf{x} - d$$

$$s = \mathbf{n} \cdot \mathbf{v} - d \Rightarrow \begin{cases} s > \epsilon, & \text{(in front of plane)} \\ s < -\epsilon, & \text{(behind plane)} \\ -\epsilon \leq s \leq \epsilon, & \text{(on plane)} \end{cases}$$

A common approach to deal with these kinds of numerical problems is to use so called \*fat\* planes

- Instead of comparing directly against zero we now compare against some epsilon value

We still can classify points when using fat planes as before:

- A point is in front of the plane if its distance is larger than epsilon
- A point is behind the plane if the distance is less than negative epsilon
- Otherwise the point must on the plane.

**We can now define a vertex to be convex if its distance is larger than epsilon. All other points are either concave or coplanar and should be handled specially!**

# Quickhull 2D: Face Merging

*Non – Convex*



*Merged*



So what do we do when we encounter a non-convex vertex?

- The rule is essentially that we want every vertex of our hull to be clearly convex
- We can enforce this rule by simply merging the left and right face **across** the non-convex vertex into a new face.
- As you can see on the slide this will remove the concavity and correct the geometrical defect.

# Quickhull 2D: Epsilon

---

$$\varepsilon = 2 \left( \max_i (abs(x_i)) + \max_i (abs(y_i)) \right) \cdot FLT\_EPSILON$$

The final question is what epsilon we should choose for our fat planes:

- The CRT defines a floating point epsilon but this does NOT take our input set into account.
- We like to define a relative tolerance which takes the size of the input object into account
- So one possible solution it to choose an epsilon relative to the sum of maximum absolute coordinates
- Note that when using `FLT_EPSILON` without scaling we would fail big time e.g. very small models (where `FLT_EPSILON` is much bigger than the entire model itself)
- Or for very large models where the epsilon is too small to make any difference at all

# Outline

---

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- **Introduce Quickhull in 3D**
- Quickhull 3D Invariants
- Implementation

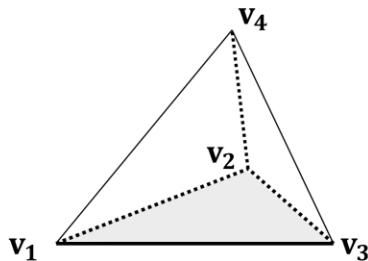
This closes the introduction of Quickhull in 2D:

- I hope you now have some first idea how the algorithm operates
- In 2D there already exist good algorithms to build convex hulls which are easy and straight forward to implement.
- So if your game is 2D I recommend using one of those.

In the remainder of the talk we will discuss how we build convex hulls in 3D

- Quickhull in 3D is very similar to the version I just showed you in 2D
- The most notable difference is the construction of the horizon and we have to deal with numerical imprecision more carefully

# Quickhull 3D: Initial Hull



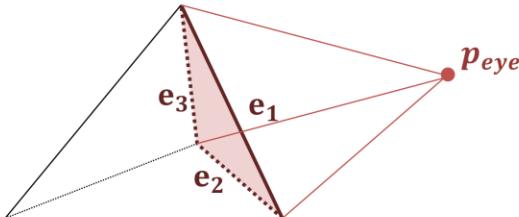
As in 2D we need to build an initial hull

- We will find the initial triangle ( $v_1, v_2, v_3$ ) as we did in 2D before
- Then we will also add the furthest point from the triangle plane (here  $v_4$ )
- In 3D the initial hull is now a tetrahedron
- After we build the initial hull we then partition the remaining points into the conflict lists of the faces of our initial hull

# Quickhull 3D: Horizon

Horizon:

$$H = \{ e_1, e_2, e_3 \}$$



We then start iteratively adding new points to the hull and grab the point with the largest distance from our conflict lists:

- This will give us the next eye point
- As in 2D we need to find the horizon again
- In 3D the horizon is a list of edges that connect visible with non-visible faces.
- And again the horizon is the boundary between the visible and non-visible part of the current hull as seen from the current eye point
- Finding the horizon is a bit more involved in 3D and we will look at it in more detail in just a second

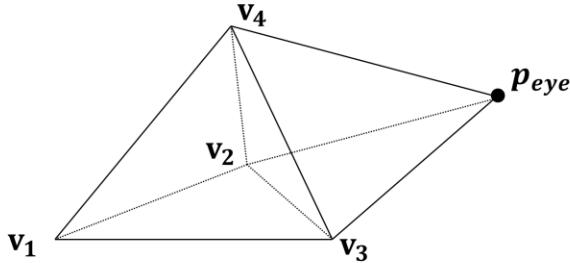
# Quickhull 3D: Add point to hull

New Faces:

$$F_1 = \{ p_{eye}, v_4, v_3 \}$$

$$F_2 = \{ p_{eye}, v_3, v_2 \}$$

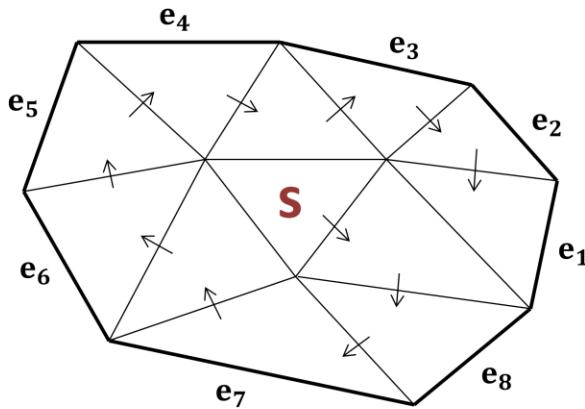
$$F_3 = \{ p_{eye}, v_2, v_4 \}$$



We then proceed with the iteration and create a new face for each horizon edge with the new eye-point

- This essentially connects the new vertex to the current hull
- Finally we partition the orphaned vertices to the new faces ( $F_1 - F_3$ )

# Quickhull 3D: Finding the Horizon



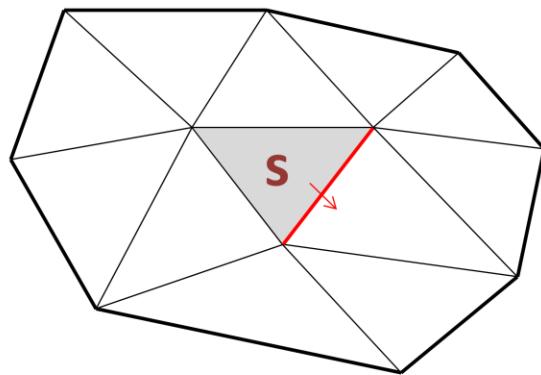
Finding the horizon in 3D is not as easy as finding two vertices as it was in 2D.

- For finding the horizon we essentially perform a DFS starting from the conflict face **S**
- At each step we cross one edge and visit a neighboring face
- If the face is visible, we cross another edge until we find a face that is not visible from the current eye point
- We store the offending edge as part of the horizon and continue the search in the previous face
- On termination we have a list of all edges defining the horizon in a closed CCW loop
- On the slide we start at the face labeled S and follow the arrows and on our way back we collect the horizon edges
- Since this is an essential operation of the hull construction let's look at this step in a bit more detail

**Explain arrows in slide!**

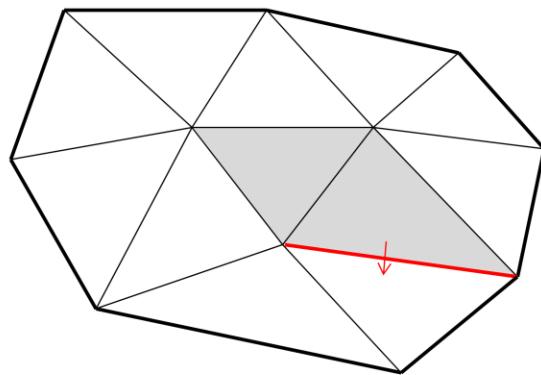
# Quickhull 3D: Find Horizon

---



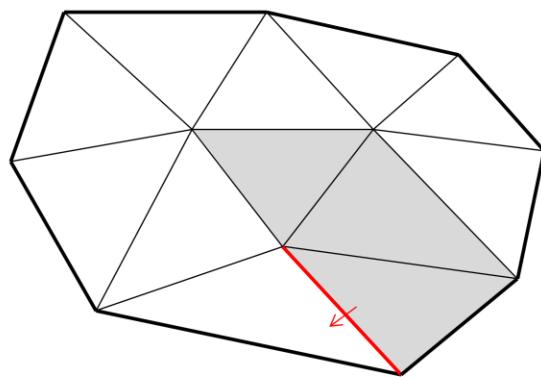
I prepared a small animation which hopefully will help to understand the horizon construction:

# Quickhull 3D: Find Horizon



- We test the visibility of the next face (and we assume here it is visible from the eye point)
- Since it is visible we continue our search and cross the next edge

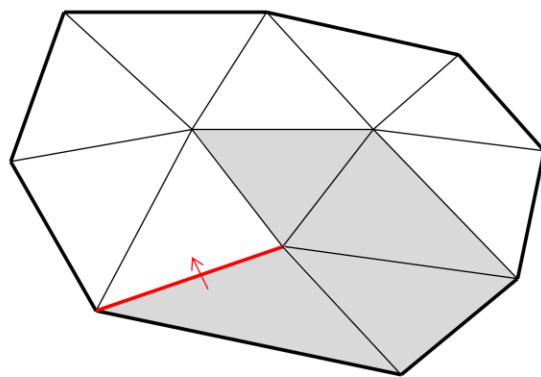
# Quickhull 3D: Find Horizon



- We test the next face and since it is visible as well we continue and cross the next edge

# Quickhull 3D: Find Horizon

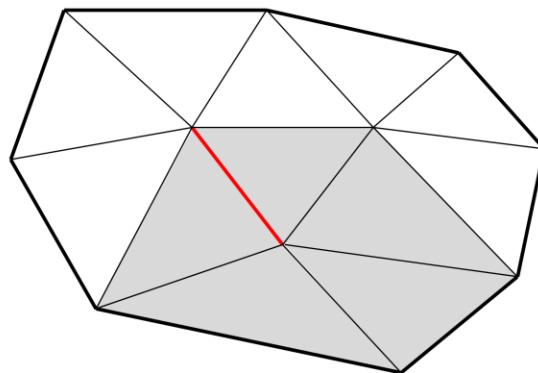
---



- We continue these tests until we cross an edge to an invisible face

# Quickhull 3D: Find Horizon

---

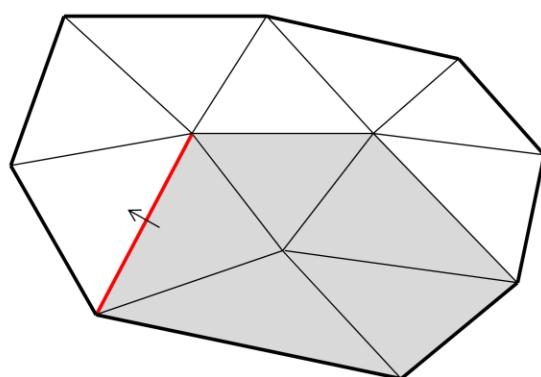


We would now cross an edge to a face we have already visited

- Whenever we visit a face we will mark them as processed
- This allows to simply test if a face was already processed and can be skipped

# Quickhull 3D: Find Horizon

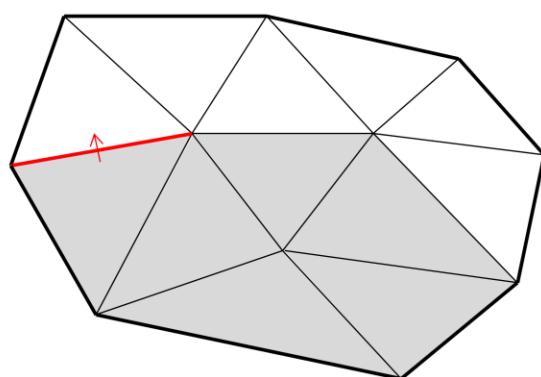
---



We continue crossing edges...

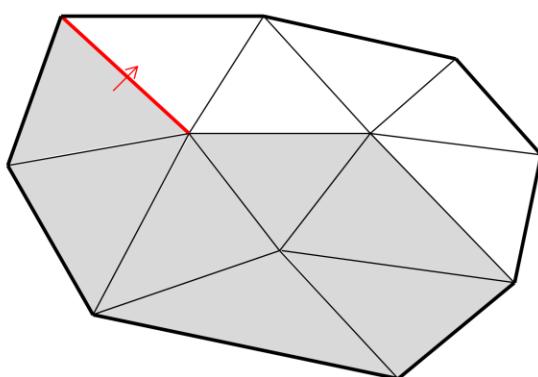
# Quickhull 3D: Find Horizon

---



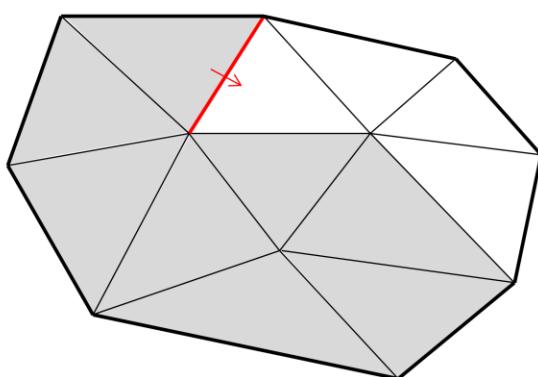
# Quickhull 3D: Find Horizon

---



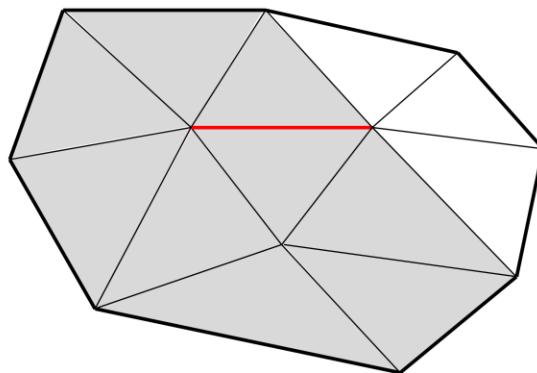
# Quickhull 3D: Find Horizon

---



# Quickhull 3D: Find Horizon

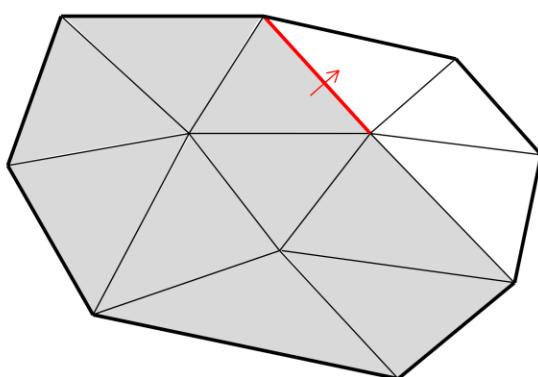
---



Again we already visited this face so we don't cross this edge as well

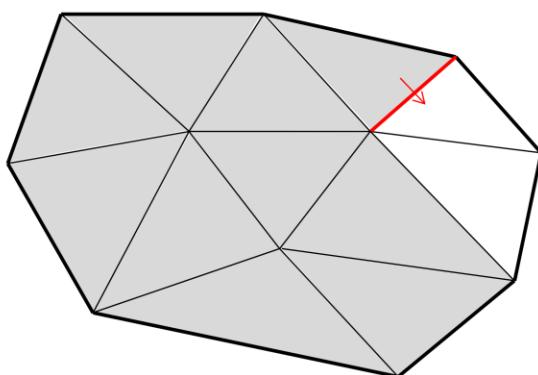
# Quickhull 3D: Find Horizon

---



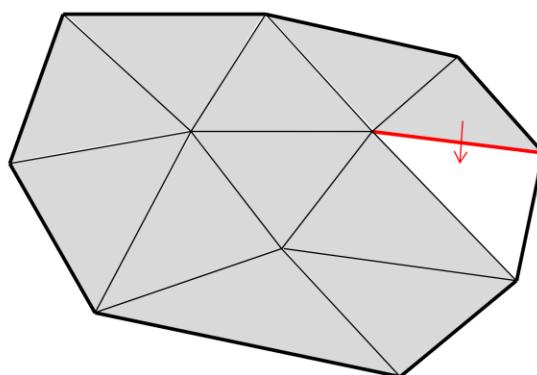
# Quickhull 3D: Find Horizon

---



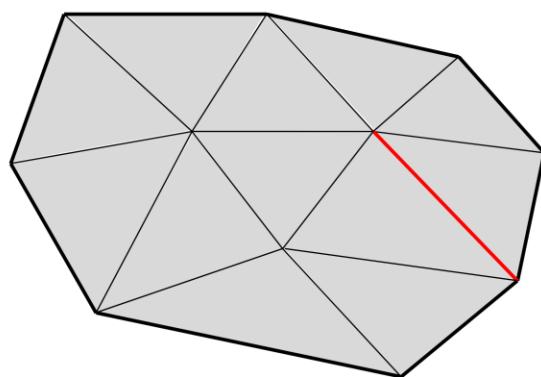
# Quickhull 3D: Find Horizon

---



# Quickhull 3D: Find Horizon

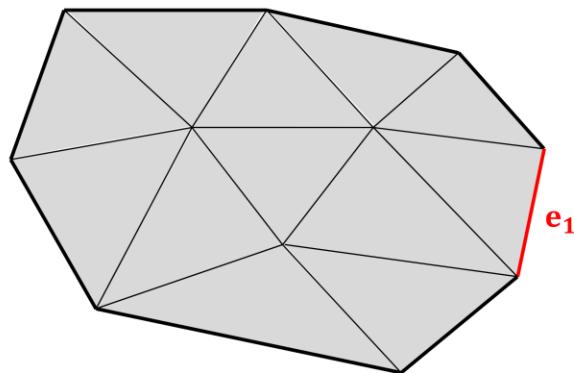
---



The next face was already visited so no need to cross here too

# Quickhull 3D: Find Horizon

---

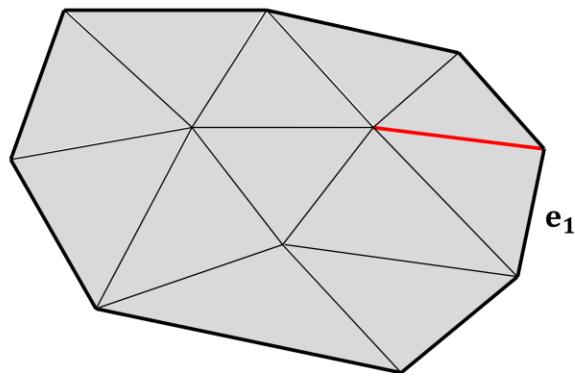


Finally we would cross the first edge that connects a visible and a non-visible face

- We add the edge to the horizon list and return to the previous face

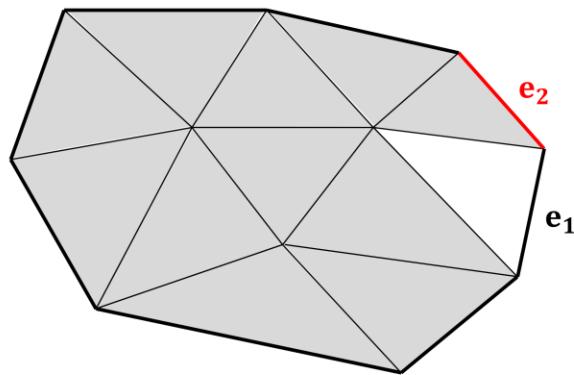
# Quickhull 3D: Find Horizon

---



We then continue with the next edge and return to the previous face

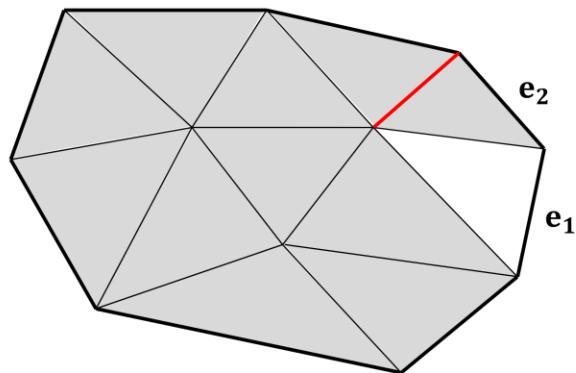
# Quickhull 3D: Find Horizon



Again we cross an edge that connects a visible and invisible face

- We save that edge and add it to our horizon and return to the previous face

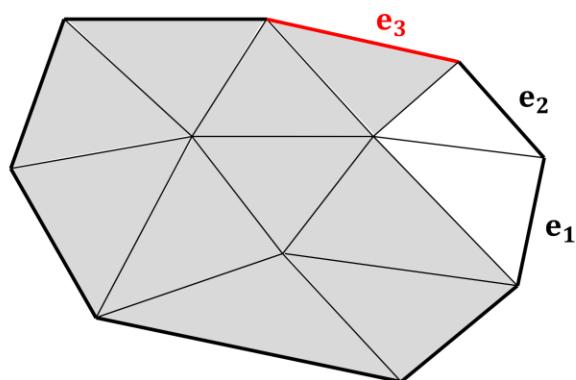
# Quickhull 3D: Find Horizon



The procedure continues and collects the horizon edges until we made our way back to the start face

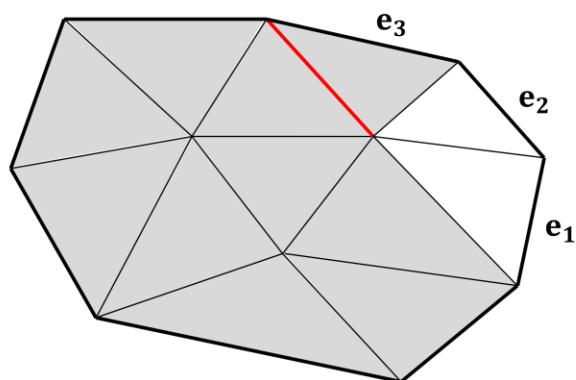
# Quickhull 3D: Find Horizon

---



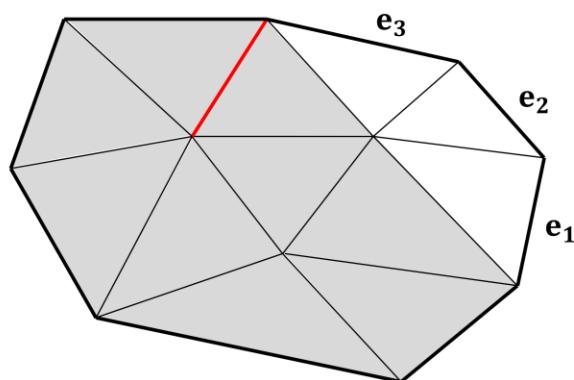
# Quickhull 3D: Find Horizon

---



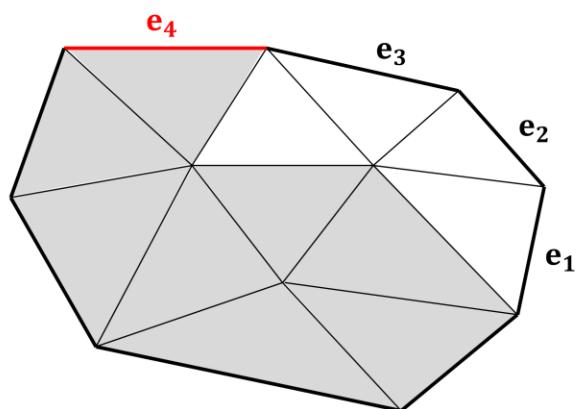
# Quickhull 3D: Find Horizon

---



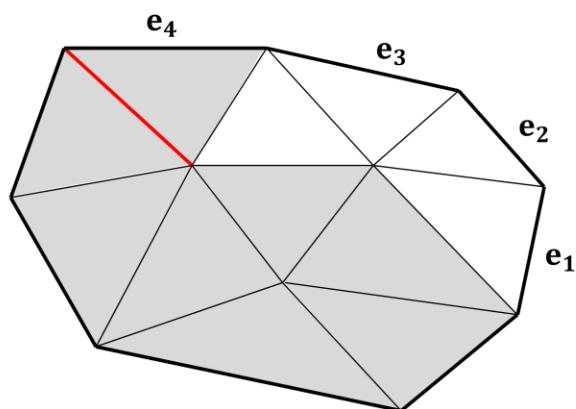
# Quickhull 3D: Find Horizon

---



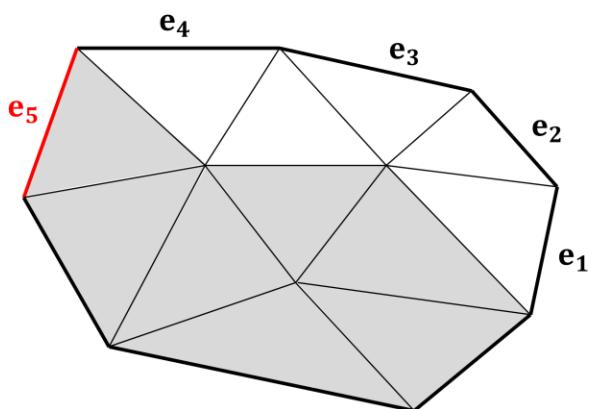
# Quickhull 3D: Find Horizon

---



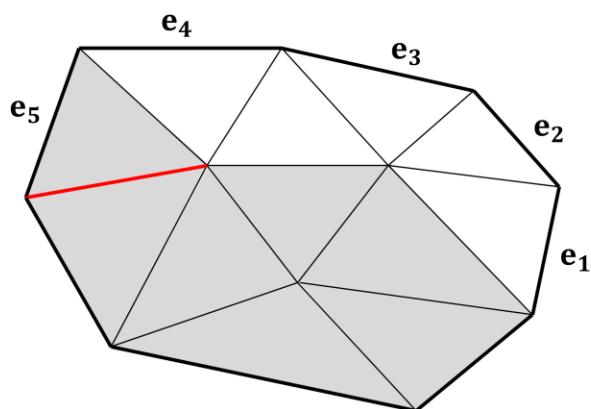
# Quickhull 3D: Find Horizon

---



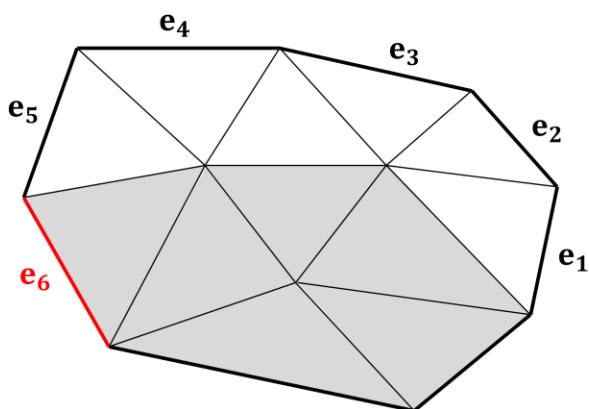
# Quickhull 3D: Find Horizon

---



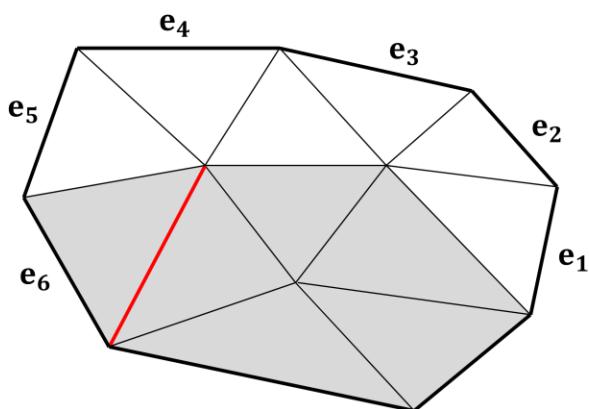
# Quickhull 3D: Find Horizon

---



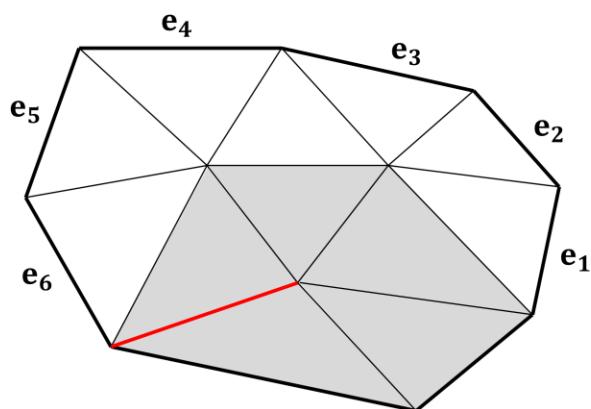
# Quickhull 3D: Find Horizon

---

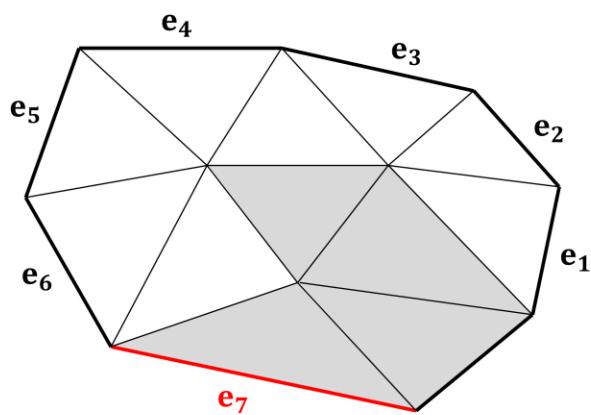


# Quickhull 3D: Find Horizon

---

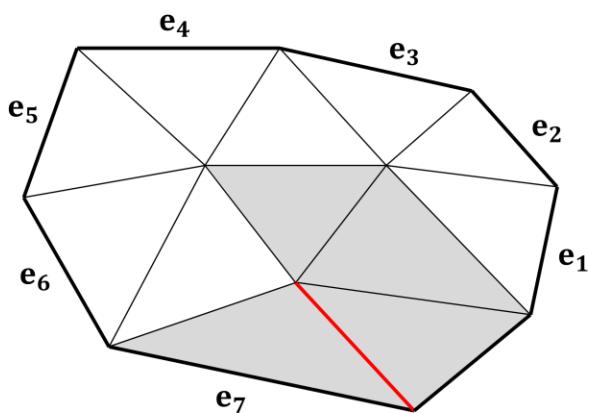


# Quickhull 3D: Find Horizon



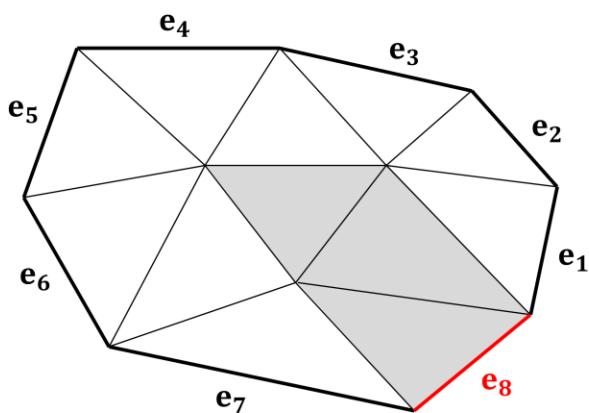
# Quickhull 3D: Find Horizon

---



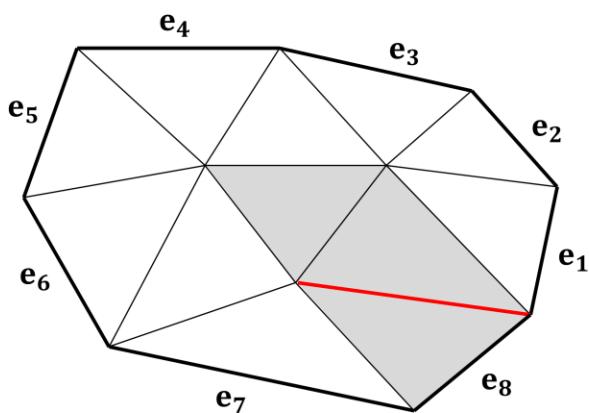
# Quickhull 3D: Find Horizon

---



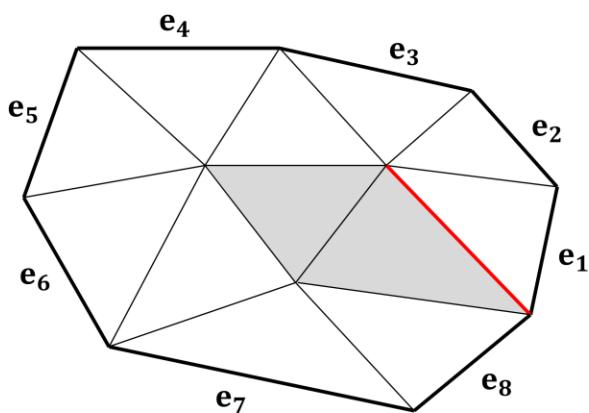
# Quickhull 3D: Find Horizon

---



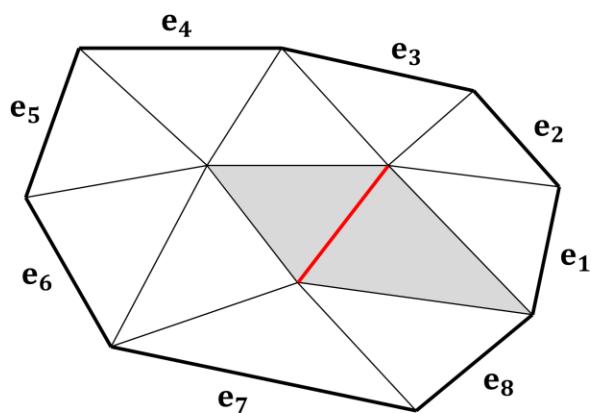
# Quickhull 3D: Find Horizon

---



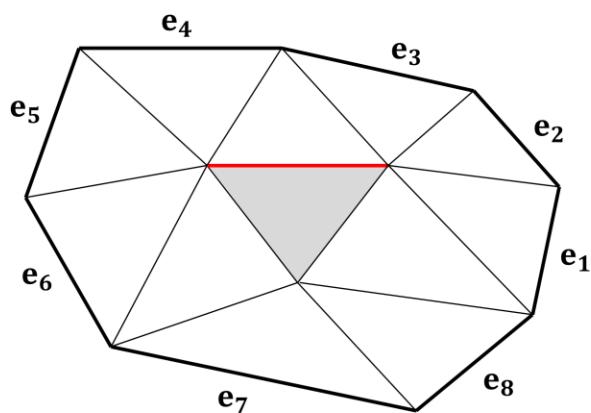
# Quickhull 3D: Find Horizon

---



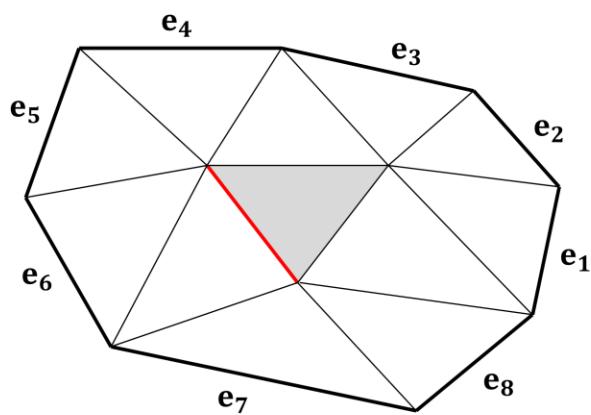
# Quickhull 3D: Find Horizon

---

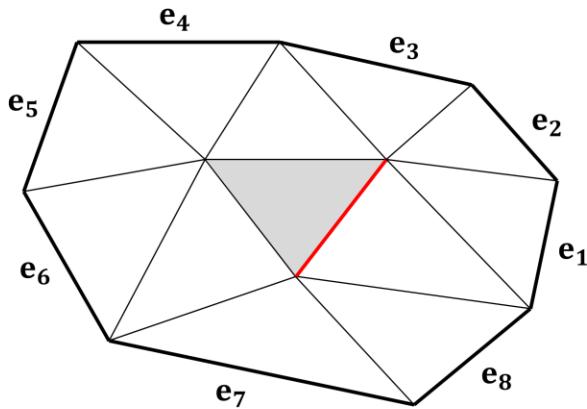


# Quickhull 3D: Find Horizon

---

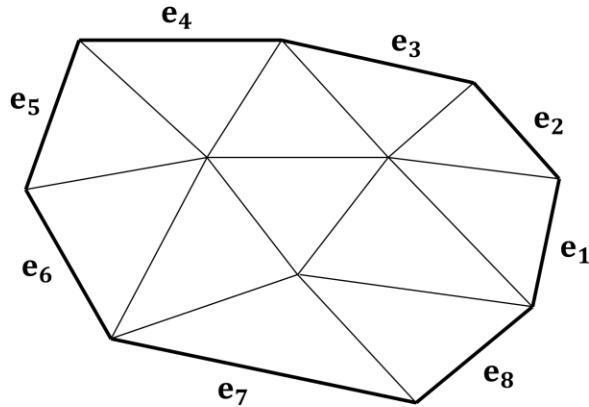


# Quickhull 3D: Find Horizon



This is edge we started with and we are done!

# Quickhull 3D: Find Horizon



On termination we have a list of all horizon edges in CCW order

- We can now simply create a new triangle face for each horizon edge in our list
- This essentially connects the new vertex to the current hull

# Outline

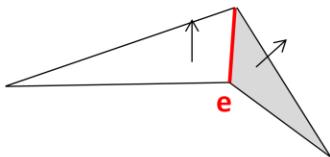
---

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- **Quickhull 3D Invariants**
- Implementation

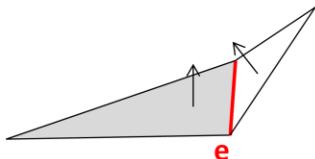
This closes the introduction of the 3D Quickhull algorithm and we continue looking into invariants

# Quickhull 3D: Invariants

Edge **e** is convex



Edge **e** is concave

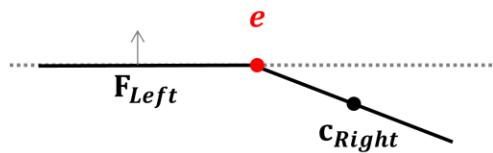


As in 2D we must maintain geometric invariants while constructing the hull:

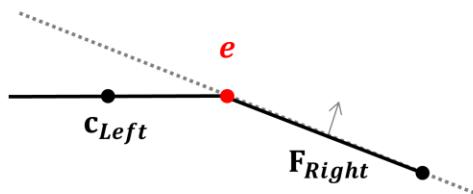
- In 3D we must now guarantee that every **edge** is convex
- And the argumentation here is basically the same as it was earlier in 2D

# Quickhull 3D: Invariants

Left Face Test



Right Face Test



We now need to define what a convex **edge** is and how we can test an **edge** for convexity:

For each edge:

- First we test if the **center** of the right face is below the left face plane
- Then we test if the **center** of the left face is below the right face plane
- If both tests are true the edge is **convex**, otherwise it must be **concave** or **coplanar**
- This is very similar to the 2D test, but we now use the center point of the face
- The face center is simply the average of the face vertices

# Quickhull 3D: Epsilon

---

$$\varepsilon = 3 \left( \max_i(\text{abs}(x_i)) + \max_i(\text{abs}(y_i)) + \max_i(\text{abs}(z_i)) \right) \cdot \text{FLT\_EPSILON}$$

As in 2D we need an epsilon to define the thickness of flat planes and we just expand our formula to 3D!

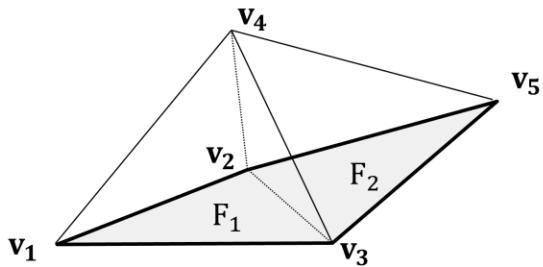
# Quickhull 3D: Merging Coplanar Faces

Coplanar Faces:

$$F_1 = \{v_1, v_2, v_3\}$$

$$F_2 = \{v_5, v_3, v_2\}$$

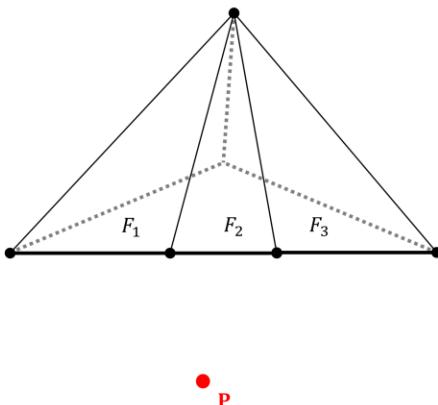
$$F_{12} = \{v_1, v_2, v_5, v_3\}$$



So when we detect a non-convex edge we now merge the two adjacent faces

- This adds another step to our iterative loop
- Let's assume face1 and face2 were non-convex in our previous example
- We would now merge face1 and face2 into a new **Polygonal** face replacing the original faces  $F_1$  and  $F_2$
- As mentioned in the beginning we are not restricting ourselves to triangle faces

# A Troublesome Hull



Situation:  $F_1$  and  $F_3$  are visible from new point  $\mathbf{P}$  and  $F_2$  is not!

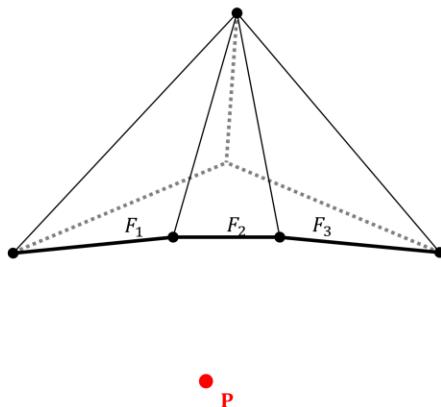
Let's now investigate the example from the original Quickhull paper:

- The message of the next slides is really to show what will happen if you DON'T merge faces and how you can run into a bunch of geometrical and topological problems!
- All you need to get here right now is that face merging is really a **critical** operation!

The situation presents itself like this: Faces  $F_1$  and  $F_3$  are visible from the new point  $\mathbf{P}$  while face  $F_2$  is not!

- Basically you can think of a simple tetrahedron here
- The front face is in the screen plane and was **not** merged into one big triangle face, but is essentially a fan of three faces
- It appears to us as if faces  $F_1 - F_3$  are coplanar
- But the bad news is that coplanar only exists in an ideal world, but not in the world of limited precision numbers!

## A Troublesome Hull (2)

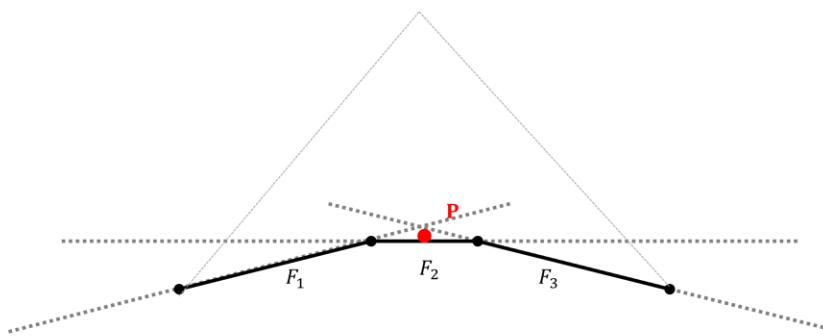


Situation:  $F_1$  and  $F_3$  are visible from new point  $\mathbf{P}$  and  $F_2$  is not!

What really happens is (and I am exaggerating the situation here a little bit) is that:

- The two center points are actually moved inside
- And the edges between  $F_1$  and  $F_2$  and also  $F_2$  and  $F_3$  are concave

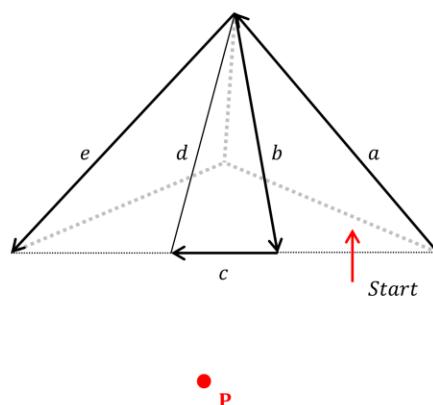
## A Troublesome Hull (2)



Now let's rotate the tetrahedron and assume we are now looking from behind point P and at the bottom :

- You can hopefully see the new point and the bottom edges of the three faces
- Due to the non-convexity the new point gets kind of locked behind the three planes that leads to this weird situation

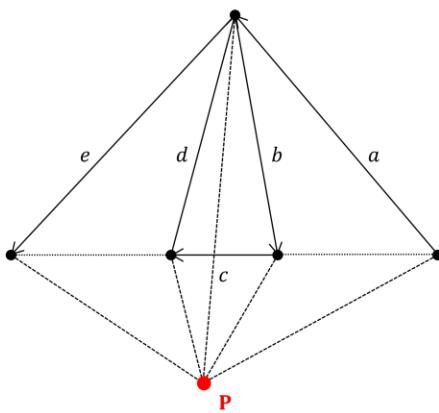
# A Pinched Horizon



The depth-first-search will yield the horizon in CCW order!

We continue and build now the horizon as we learned earlier using the DFS  
- The results in five horizon edges a, b, c, d, and e as sketched on the slide

# New Faces

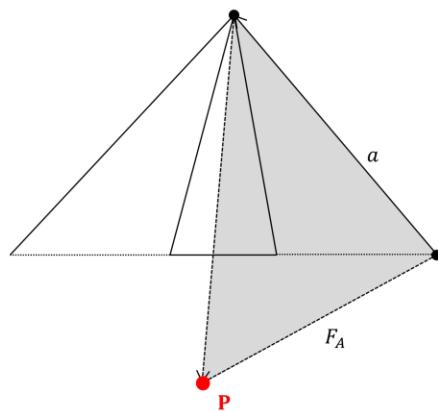


We replace  $F_1$  and  $F_3$  with 5 new faces for each edge a-e

Next we replace  $F_1$  and  $F_3$  with 5 new faces for each edge a-e

- Since the outcome of the operation is pretty interesting let's look at each of the new faces individually

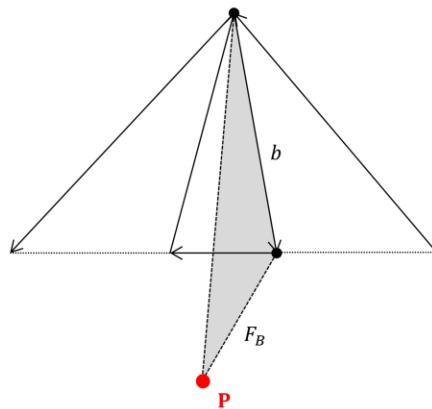
## Face A



We build a face for edge a

- Note how this face partially overlaps face2 which is still on the hull

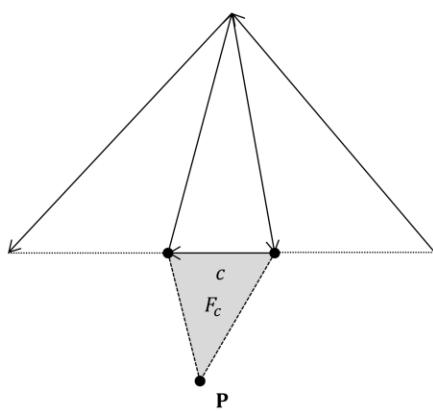
## Face B



We build a face for edge b

- Note that this face has flipped (CW) orientation and also shares an edge with FA

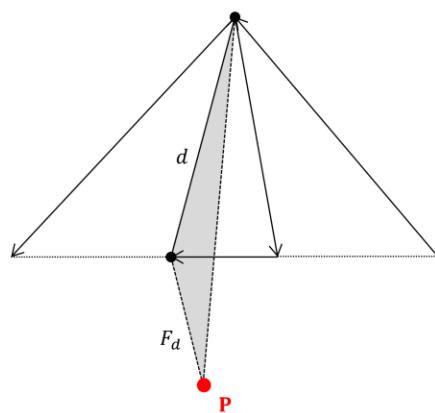
# Face C



We build a face for edge  $c$

- Hooray, nothing wrong here

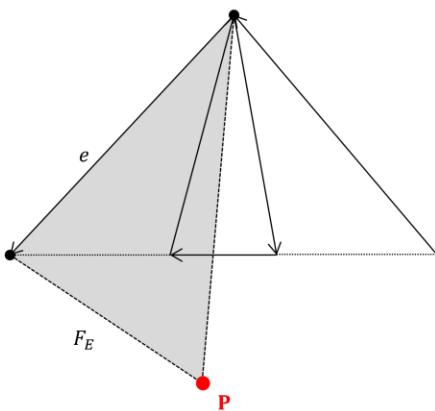
## Face D



We build a face for edge  $d$

- Note that this face has flipped (CW) orientation again and also shares an edge with FA and FB

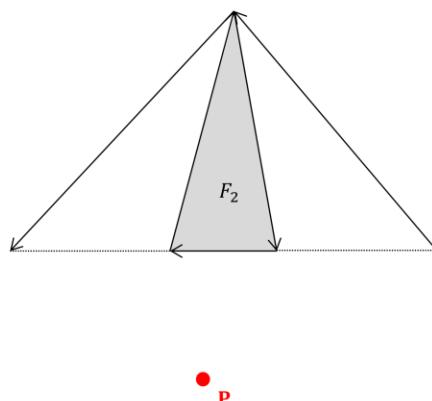
# Face E



We build a face for edge  $e$

- We now have four faces sharing the same edge and also partially overlapping each other

## Face 2



$F_2$  is still on the hull as well because it cannot be seen from the new point!

- Due to numerical imprecision face2 was still identified as visible from P and will remain on the hull

# Errors

## ➤ Geometrical Errors

- Two faces are flipped upside-down ( $F_B, F_D$ )

## ➤ Topological Errors

- Four faces share the same edge ( $F_A, F_B, F_D, F_E$ )

What happened here is that we introduced a couple of severe errors because our hull was not in an healthy state when we started adding the new point.

As a result the new faces are violating a bunch geometrical and topological invariants:

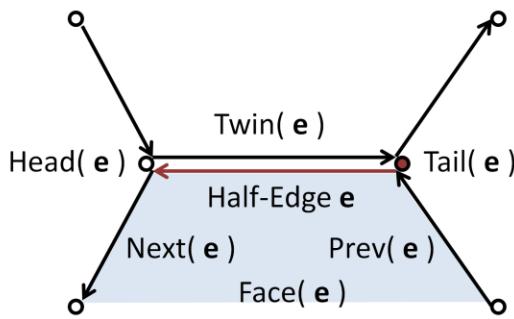
- Two faces are flipped upside down (that means the normals are pointing inside)
- Four faces share the same edge (which make them partially and fully overlapping each other)

As you can imagine repairing those errors would become pretty involved.

The good news are that I haven't run into any of the described issues when properly merging faces during the hull construction.

So hopefully you can see that it essential for a robust implementation to maintain a healthy hull during construction.

# Half-Edge Mesh



We haven't talked about a data structure for convex polyhedra yet. So before we start looking into face merging in more detail, let's talk about a possible data structure first:

- Obviously there are many ways to describe a convex polyhedron
- A common data structure is the so called Half-Edge data structure which is an edge centric mesh representation
- The half edge data structure makes it easy to iterate edges of a face and to access neighboring faces

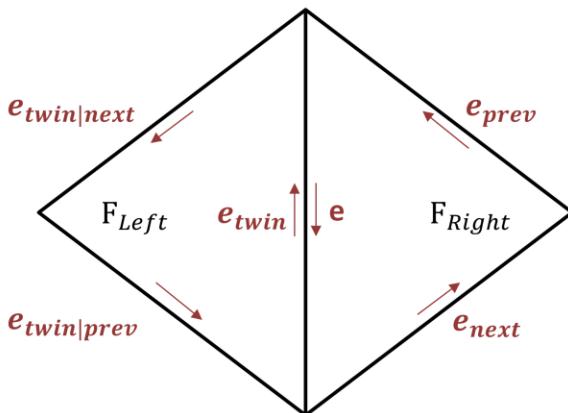
For each face we store:

- A half-edge that defines the entry into a circular list around the face

For each edge we store:

- The previous and next edge that build the circular list around the face
- A twin edge to cross over to the adjacent face
- And the tail vertex of the edge
- Note that we don't need to store the head vertex since it is simple the tail vertex of the twin edge

# Face Merging



Set edge reference:  
 $e \rightarrow \text{face} \rightarrow \text{edge} = e \rightarrow \text{prev}$

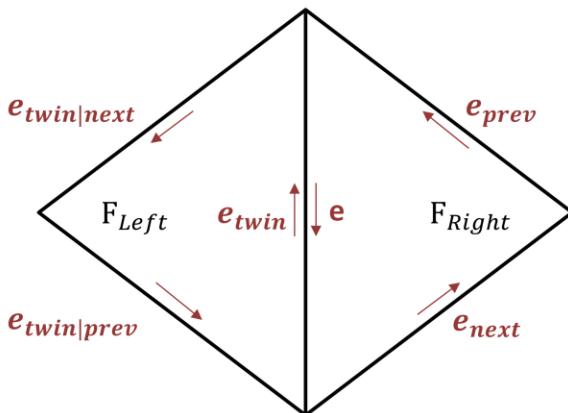
Absorb face:  
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{face} = e \rightarrow \text{face}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{face} = e \rightarrow \text{face}$

Link edges:  
 $e \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{twin} \rightarrow \text{next}$   
 $e \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{twin} \rightarrow \text{prev}$   
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{next}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{prev}$

We learned that face merging is the essential operation to maintain a healthy hull.

- Here is an example how to merge two faces using the half-edge data structure.
- The situation is that we are about to merge the left into the right face
- This means that edge e and its twin will go away and also the left face

# Face Merging



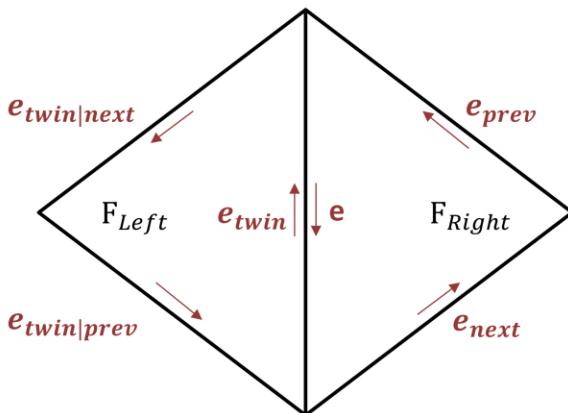
**Set edge reference:**  
 $e \rightarrow \text{face} \rightarrow \text{edge} = e \rightarrow \text{prev}$

**Absorb face:**  
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{face} = e \rightarrow \text{face}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{face} = e \rightarrow \text{face}$

**Link edges:**  
 $e \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{twin} \rightarrow \text{next}$   
 $e \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{twin} \rightarrow \text{prev}$   
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{next}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{prev}$

First we make sure that the absorbing right face does not reference the edge we are about to delete (e.g. we use edge->prev here)

# Face Merging



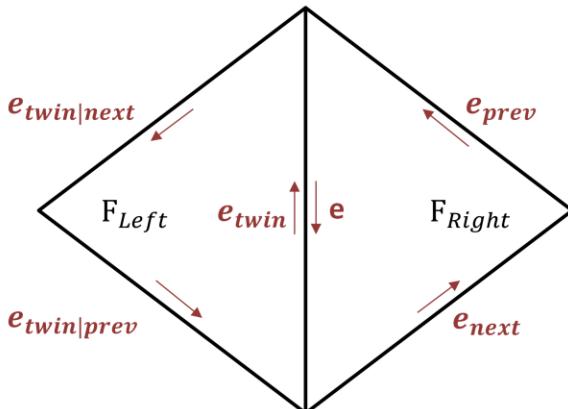
Set edge reference:  
 $e \rightarrow \text{face} \rightarrow \text{edge} = e \rightarrow \text{prev}$

**Absorb face:**  
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{face} = e \rightarrow \text{face}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{face} = e \rightarrow \text{face}$

Link edges:  
 $e \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{twin} \rightarrow \text{next}$   
 $e \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{twin} \rightarrow \text{prev}$   
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{next}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{prev}$

Next we must make sure that all edges of the absorbed left face will now reference the right face as their owner

# Face Merging



Set edge reference:  
 $e \rightarrow \text{face} \rightarrow \text{edge} = e \rightarrow \text{prev}$

Absorb face:  
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{face} = e \rightarrow \text{face}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{face} = e \rightarrow \text{face}$

Link edges:  
 $e \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{twin} \rightarrow \text{next}$   
 $e \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{twin} \rightarrow \text{prev}$   
 $e \rightarrow \text{twin} \rightarrow \text{prev} \rightarrow \text{next} = e \rightarrow \text{next}$   
 $e \rightarrow \text{twin} \rightarrow \text{next} \rightarrow \text{prev} = e \rightarrow \text{prev}$

Finally we need to connect the incoming and outgoing edges

- Also note that we can access all necessary data for the merge operation just from the shared edge between the two adjacent faces we are about to merge

# Face Merging: Newell Plane

- The result two merged faces is a polygon with its vertices **not** quite coplanar
- Remember that we are in the world of limited precision numbers
- The Newell algorithm builds a best fit plane in the least square sense which minimizes the distance of the vertices

The result of two of merged faces is a polygonal face with its vertices **not** quite coplanar

- Remember that we are in the world of limited precision numbers
- The Newell algorithm builds a best fit plane in the least square sense which minimizes the distance of the vertices

# Face Merging: Newell Plane

$$n_x = \sum_{i=1}^N (y_i - y_{i+1}) \cdot (z_i + z_{i+1})$$

$$n_y = \sum_{i=1}^N (z_i - z_{i+1}) \cdot (x_i + x_{i+1})$$

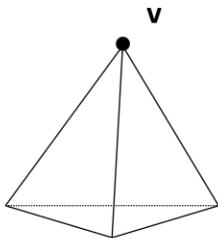
$$n_z = \sum_{i=1}^N (x_i - x_{i+1}) \cdot (y_i + y_{i+1})$$

$$\mathbf{p} = \frac{1}{N} \sum_{i=1}^N \mathbf{v}_i$$

I added some formulas for completeness and convenience, but going into detail here would get us off topic.

- Both Gino's and Christer's books cover Newell planes and I also added some additional links in the references

# Topological Invariants



Each vertex must have at least three adjacent faces

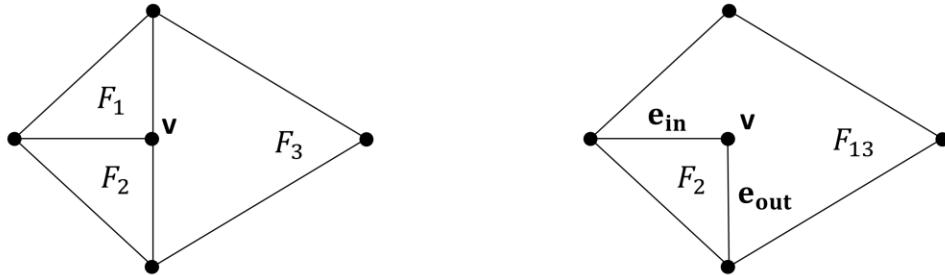
Now let's look at one important topological invariant of a convex hull.

- The most important one for us is that each vertex must have at least three adjacent faces
- We call a vertex redundant if it does not satisfy this invariant

Of course there are other topological invariants, like:

- Each face must have at least three neighbors
- Each edge is shared exactly between two faces
- Each face of a convex polyhedron must be a convex polygon
- But as it turns out we don't need to deal with those directly

# Fixing Topological Errors (1)



When merging faces we might violate topological invariants and need to fix those:

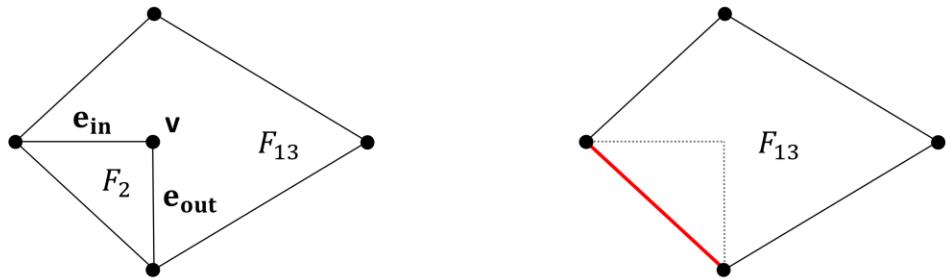
Consider the merge sequence in the above picture and how it can lead to topological errors:

- We merge face1 into face3

This creates a couple of problems:

- Vertex v has now only two adjacent faces and has become redundant
- Face2 has only two neighbors
- Face13 is not convex

# Fixing Topological Errors (2)

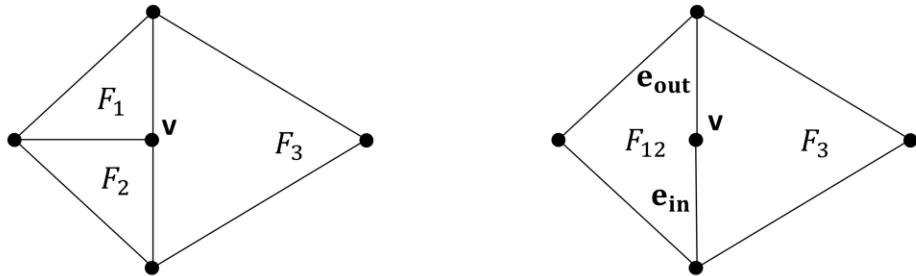


We detect this error by checking the adjacent faces of the in- and outgoing edges ->  
Both edges point to face2

- Since face2 is a **triangle** we will not connect the in- and outgoing edge, but use the non-shared edge instead
- Note that face2 is redundant as well since all vertices are contained in face13 and can be deleted
- Vertex  $v$  has also become obsolete and will be deleted as well

**The trick here is that when we merge two faces we check for this error and fix it immediately!**

# Fixing Topological Errors (3)



Let's look at the previous example again.

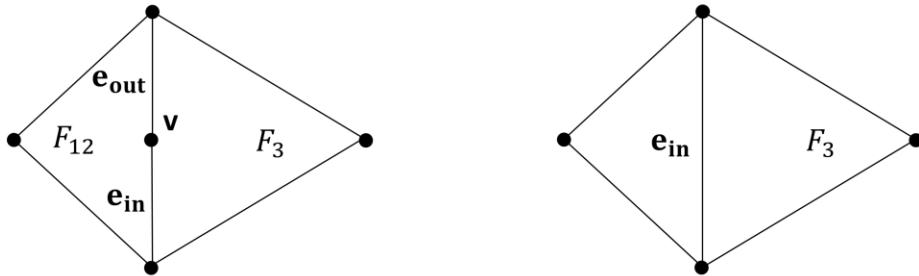
Consider now the slightly different merge sequence:

- We now merge face2 into face1

Again:

- Vertex v has now only two adjacent faces and has become redundant
- Besides this I don't notice any other problem in this case

# Fixing Topological Errors (4)

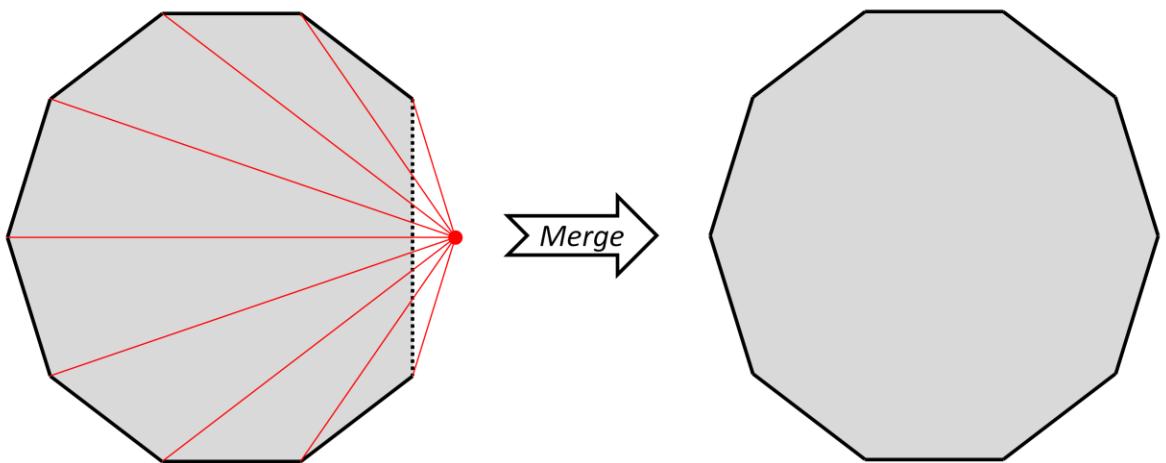


We detect this error again by checking the adjacent faces of the in- and outgoing edges -> Both edges point to face3 here

- Since face3 has now more than three vertices we cannot apply the same fixing strategy as before
- Instead we simply extend the incoming edge to the next vertex and delete the outgoing edge
- Again vertex  $v$  has become obsolete and should be deleted as well

**Again when we merge faces we check for this error and fix it immediately!**

# Large Merge Cycles



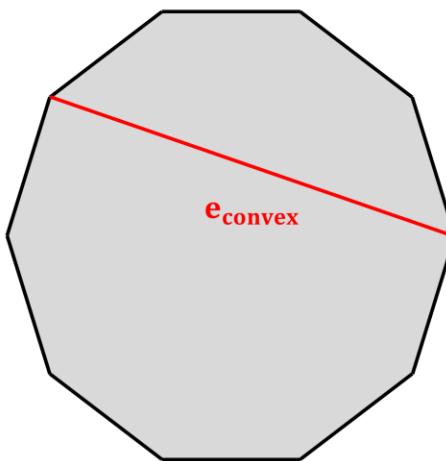
Another problem you will encounter when merging faces is that you might a large number of new faces that should be all merged:

- Let's call this a merge cycle

Imagine we are building the convex hull of a cylinder and we are about to add the final vertex of the top face:

- This vertex of course presents itself as in the same plane as the other vertices.
- In this situation we create many new faces which are roughly coplanar and need to be merged
- Ideally we would like to merge all new faces into one face as shown on the right hand side

# Large Merge Cycles (2)



The problem is now that we practically merge one face after the other:

- Whenever we merge two faces we rebuild the face plane.
- Rebuilding the face can jiggle the plane and an edge between two faces can become temporarily convex and prevent us from merging the whole cycle
- In the worst case this can introduce concave faces which are now **NOT** merged properly

Here are some ideas how to address this:

- The faces with the largest area should be the most stable w.r.t. the orientation of the face plane. So merging into the largest faces first reduces jiggle. Think of merging the little sister into the big daddy
- You can also introduce an absolute tolerance to increase your merge radius and make your merge cycle less sensitive for these situations.
- This is basically how I handle this problem at the moment since for physics we want as large faces as possible for stability reasons and do not aim for the tightest hull.
- If you are working with collision margins it is probably a good idea to make this absolute tolerance a small percentage of that margin

# Large Merge Cycles (2)



Another idea to deal with this problem is to **NOT** rebuild the face planes at all when merging to faces:

- For both faces you have your best fit plane and the vertices (which I tried to sketch on the slide)
- We can now compute the absolute distance of the right face's vertices to the left face's plane and vice versa
- Then we simply keep the face plane that minimizes the distance instead of rebuilding it

# Face Merging Strategies

- At each iteration we assume to start with a correct hull
- When adding the vertex we need to inspect all new faces for possible defects
- An easy strategy is to simply iterate all new faces and inspect their edges until we repaired all possible defects

Let's close this part how we could use face merging to deal with defect hulls:

- At each iteration the basic assumption is to start with a healthy hull
- When adding the vertex we need to inspect all new faces for possible defects at their edges between each other and at the horizon
- These are the weak spots where might have introduced new errors
- An easy strategy is to iterate all new faces and repair each edge one by one

# Outline

---

- Introduction
- Introduce Quickhull in 2D
- Quickhull 2D Invariants
- Introduce Quickhull in 3D
- Quickhull 3D Invariants
- **Implementation**

This closes the theory and in the remainder of this talk I like to share some quick tips about a possible implementation

# Memory Bounds

- Worst case is that all input vertices are on the hull
- The number of vertices is then at most  $V = N$
- The number of edges is then at most  $E = 3N - 6$
- The number of faces is then at most  $F = 2N - 4$
- Verify with Euler's formula:  $V - E + F = 2$   
 $N - (3N - 6) + (2N - 4) = N - 3N + 6 + 2N - 4 = 2$  (ok)

The major performance pitfall is bad memory management of the half-edge data structure:

- The convex hull for  $N$  vertices is bounded
- Worst case is that all input vertices are on the hull:
- The number of vertices is then at most  $V = N$
- The number of edges is then at most  $E = 3N - 6$
- The number of faces is then at most  $F = 2N - 4$
- Test with Euler's formula:  $V - E + F = 2$

In our implementation we can pre-allocate one buffer for vertices, half-edges, and faces and manage this buffer in a free list

- Ideally we will just have one big allocation per hull construction!
- This becomes especially important if you plan to compute convex hulls at runtime (e.g. for destruction)

Here are some practical details:

- Don't forget that you need to allocate half-edges (which is twice the number of edges)
- We also need to account for temporary allocations (e.g. horizon faces)
- In practice I just double the buffer size

# Implementation : Half-Edge Mesh (1)

```
struct qhVertex
{
    qhVertex * Prev;
    qhVertex * Next;

    // Optional
    qhHalfEdge* Edge;

    qhVector3 Position;
};
```

Let's start with the vertex structure.

- I am using an intrusive list to store the vertices
- The vertices are either on the hull or in a conflict list
- So obviously we have to include the list pointers here.
- We can optionally also store an edge leaving the vertex.
- This is not needed for constructing the hull, but it can be useful for post-processing if you like e.g. to iterate all adjacent faces of the vertex
- Of course we also need to store the position of a vertex

# Implementation: Half-Edge Mesh (2)

```
struct qhHalfEdge
{
    qhVertex* Tail;

    qhHalfEdge* Prev;
    qhHalfEdge* Next;
    qhHalfEdge* Twin;

    qhFace* Face;
};
```

Now let's have a quick look how we can potentially implement a half-edge:

- As we would expect this definition maps pretty directly to a possible data structure
- We store a pointer to the tail vertex of the edge
- The half-edges build a circular list around the face so we also need to store the list pointers here
- And of course we also store the twin edge to cross over to the adjacent face
- Finally we also keep a reference to the parent face of the edge

# Implementation : Half-Edge Mesh (3)

```
struct qhFace
{
    qhFace* Prev;
    qhFace* Next;

    qhHalfEdge* Edge;
    qhList<qhVertex> ConflictList;
};
```

Finally the face structure:

- I am also using an intrusive list here, so we have to include the list pointers again
- And of course there is a pointer to the first edge starting the circular list around the face
- Finally it is possibly a good idea to store our conflict list here as well

# Implementation: Construction

```
void qhConvex::Construct( const qhArray< qhVector3 >& Vertices )
{
    if ( !BuildInitialHull( Vertices ) )
        return;

    qhVertex* Vertex = NextConflictVertex();
    while ( Vertex != NULL )
    {
        AddVertexToHull( Vertex );
        Vertex = NextConflictVertex();
    }
};
```

Finally some high-level code examples to give you an idea of a possible implementation:

- Assume we have some qhConvex class to store the hull after construction
- This snippet shows the top level construction function

Explain a bit...

# Implementation: Iteration

```
void qhConvex::AddVertex( const qhVertex* Vertex )
{
    qhArray< qhHalfEdge* > Horizon;
    BuildHorizon( Horizon );

    qhArray< qhFace* > NewFaces;
    BuildNewFaces( NewFaces, Horizon );
    MergeFaces( NewFaces );

    ResolveOrphans( NewFaces );
};
```

The code snippets shows the iterative AddVertex() function

- We add new points until our conflict lists are empty

Explain a bit...

# Dude, where is the code?

---

- QHull (C/C++)
- [www.qhull.org](http://www.qhull.org)
- Quickhull3D (Java)
- <http://www.cs.ubc.ca/~lloyd/java/quickhull3d.html>

So you want to implement Qhull yourself and I talked now for nearly an hour and there is no code!

- Luckily there is a beautiful open-source implementation in JAVA which you can use to start
- I also recommend looking at the original Qhull implementation which is not only a great implementation, but also full of gems of computational geometry!!!

This closes the talk and hopefully my presentation will help you to understand and implement a robust convex hull builder!

Thank you!

# Thanks!

---

- Thanks to Valve for giving me the opportunity to present here today
- Thanks to Paul, Steve, Jeff, Gurjeet, Paul, Shannon, Sergiy, Erin and Anoush for rehearsing
- Special thanks to Randy Gaul for reading the presentation at an early stage and helping to iterate on the content early and often
- Special thanks to Bruce Dawson for helping with the numerical problems in Quickhull and also providing all kinds of other valuable feedback
- Special thanks to John Lloyd for sharing his beautiful JAVA Quickhull implementation and making it open source

Before I close I like to thank a bunch of people!

- Thanks to Valve for giving me permission to present to you today
- Thanks to Paul, Steve, Jeff and Anoush for spending time and rehearse this presentation with me
- Thanks to Randy for reading the presentation several times at an early stage and providing valuable feedback
- Special thanks to Bruce Dawson for helping with the numerical problems in Quickhull and also providing all kinds of other valuable feedback
- Special thanks to John Lloyd for sharing his beautiful JAVA Quickhull implementation and making it open source

# Questions

---

## More Cool Stuff

---

[www.steamdevdays.com](http://www.steamdevdays.com)

If you liked this talk and if you want to see more cool Valve presentations please visit our website!

# References

---

## Quickhull:

- C. Barber/D. Dobkin: "The Quickhull Algorithm for Convex Hulls"
- R. Seidel: "Small Dimensional Linear Programming and Convex Hulls Made Easy"
- J. Smith: "Quickhull 3D"

## Books:

- G. v. d. Bergen: "Collision Detection in Interactive 3D Environments"
- C. Ericson: "Real-time Collision Detection"
- J. O'Rourke: "Computational Geometry in C"
- M. de Berg: "Computational Geometry"
- E. Langetepe: "Geometric Data Structures for Computer Graphics"

# References

---

## **Related Publications:**

- G. Rhodes: "Computational Geometry" (Half-Edge Mesh)
- M. McGuire: "The Half-Edge Data Structure" (Half-Edge Mesh)
- D. Gordon: "Computing the Plane Equation of a Polygon" (Newell Plane)
- E. Catto: "A Troublesome Triangle" (Newell Plane)
- S. Melax: "Interaction with 3D Geometry" (StanHull)

## **Floating Point Articles:**

- B. Dawson: <http://randomascii.wordpress.com/category/floating-point/>
- C. Ericson: <http://realtimecollisiondetection.net/pubs/Tolerances/>