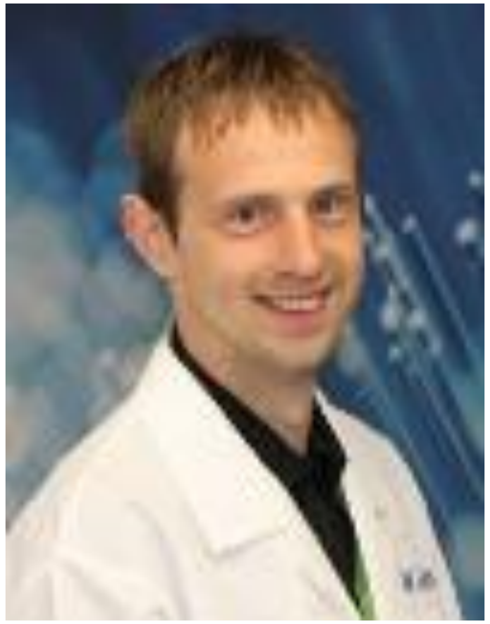
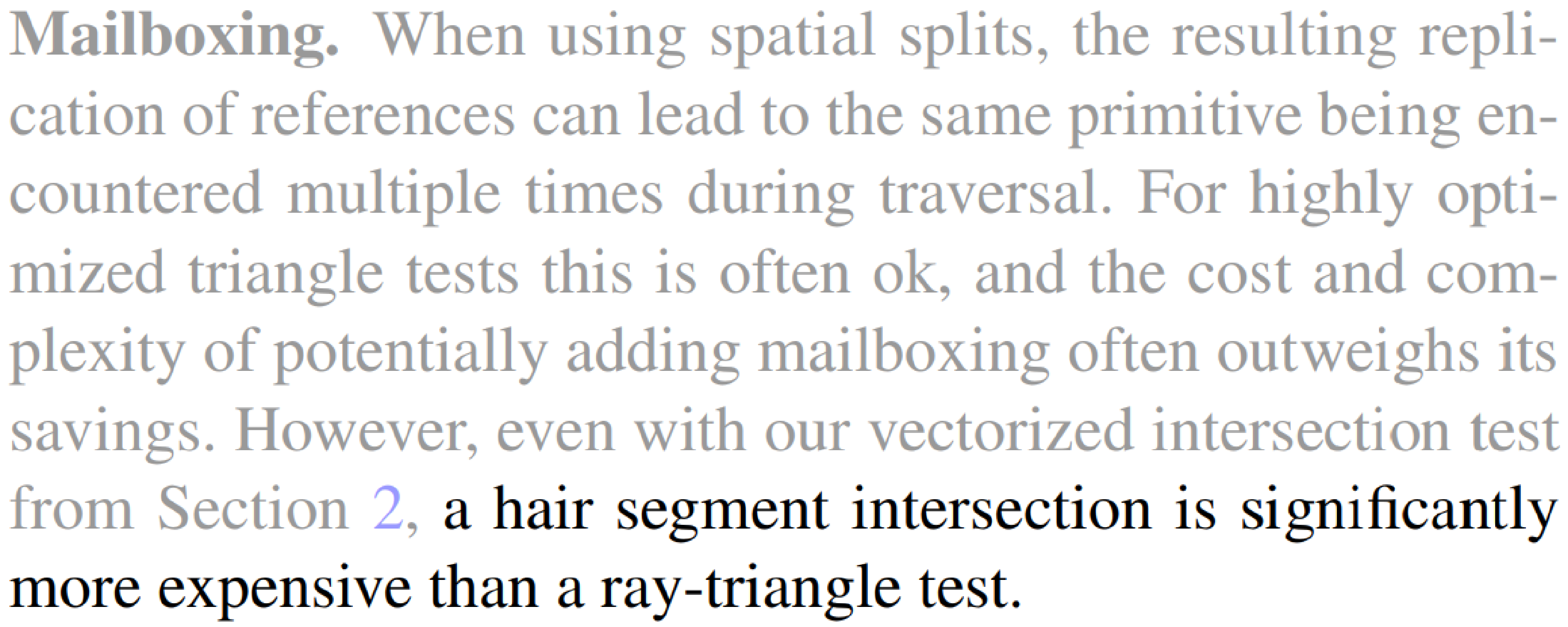
Phantom Ray-Hair Intersector

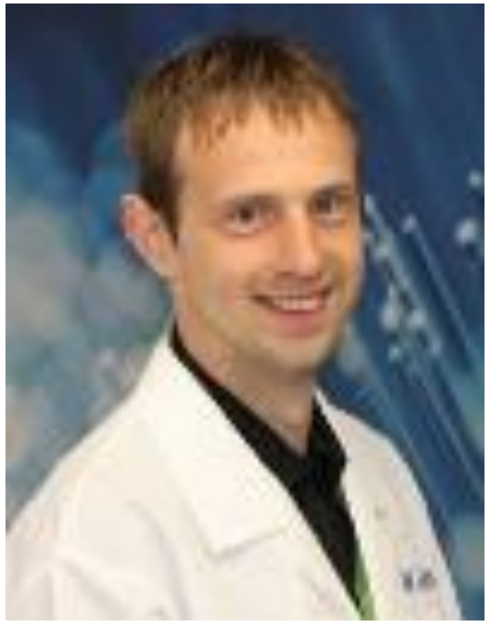
Alex Reshetov
David Luebke
NVIDIA

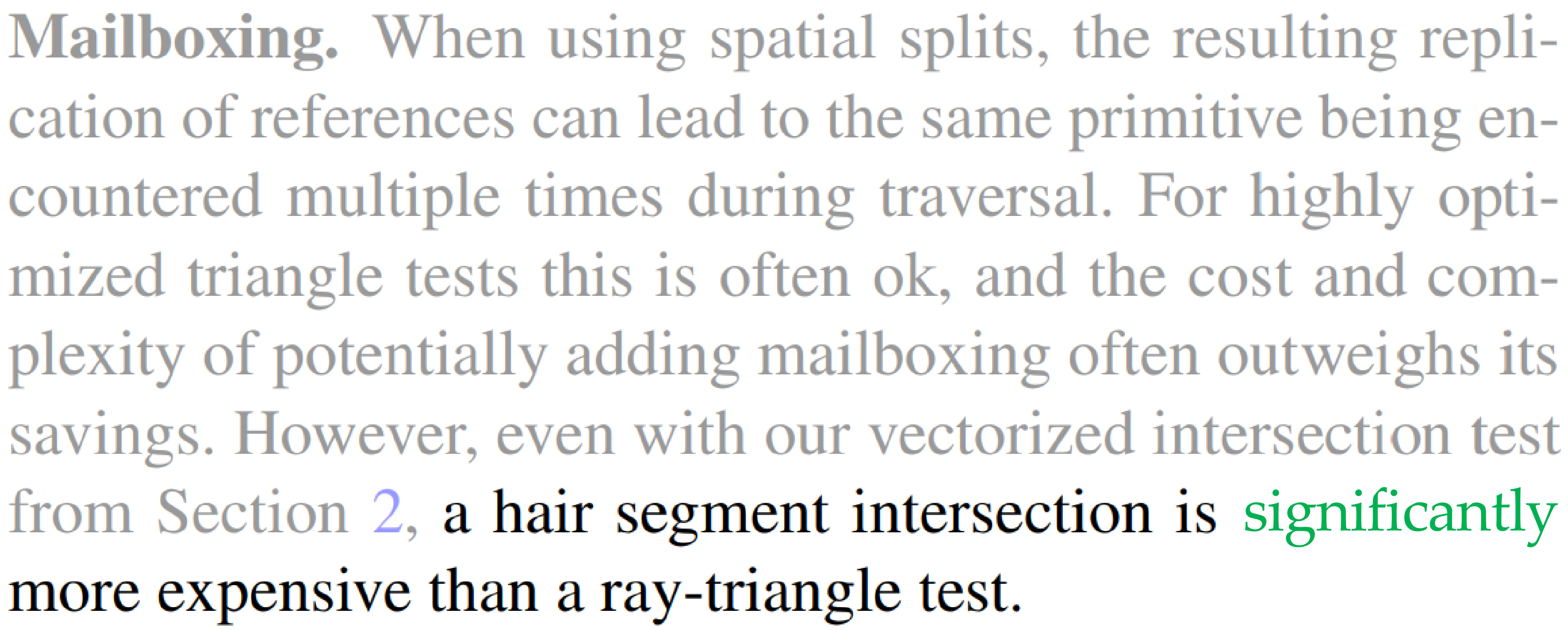




Mailboxing. When using spatial splits, the resulting replication of references can lead to the same primitive being encountered multiple times during traversal. For highly optimized triangle tests this is often ok, and the cost and complexity of potentially adding mailboxing often outweighs its savings. However, even with our vectorized intersection test from Section 2, a hair segment intersection is significantly more expensive than a ray-triangle test.

Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur
Sven Woop, Carsten Benthin, Ingo Wald, Gregory S. Johnson, and Eric Tabellion
High Performance Graphics 2014





Mailboxing. When using spatial splits, the resulting replication of references can lead to the same primitive being encountered multiple times during traversal. For highly optimized triangle tests this is often ok, and the cost and complexity of potentially adding mailboxing often outweighs its savings. However, even with our vectorized intersection test from Section 2, a hair segment intersection is significantly more expensive than a ray-triangle test.

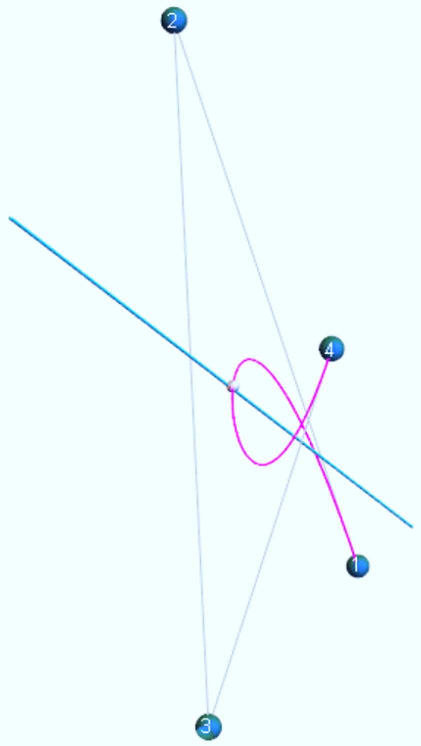
Exploiting Local Orientation Similarity for Efficient Ray Traversal of Hair and Fur
Sven Woop, Carsten Benthin, Ingo Wald, Gregory S. Johnson, and Eric Tabellion
High Performance Graphics 2014

Phantom Ray-Hair Intersector

Alex Reshetov
David Luebke
NVIDIA

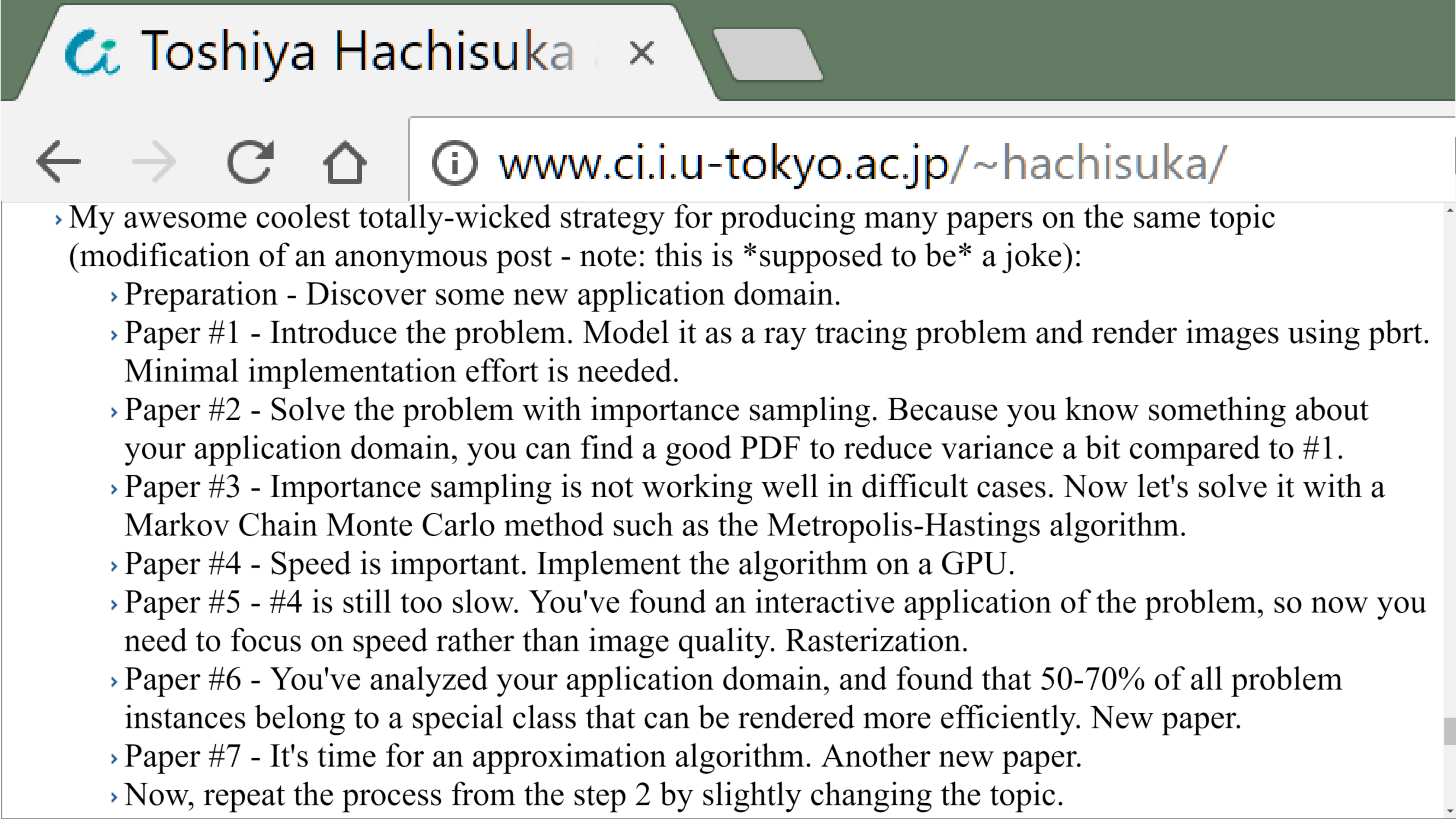
Exploiting Budan-Fourier and Vincent's Theorems for Ray Tracing 3D Bézier Curves

Alex Reshetov
NVIDIA

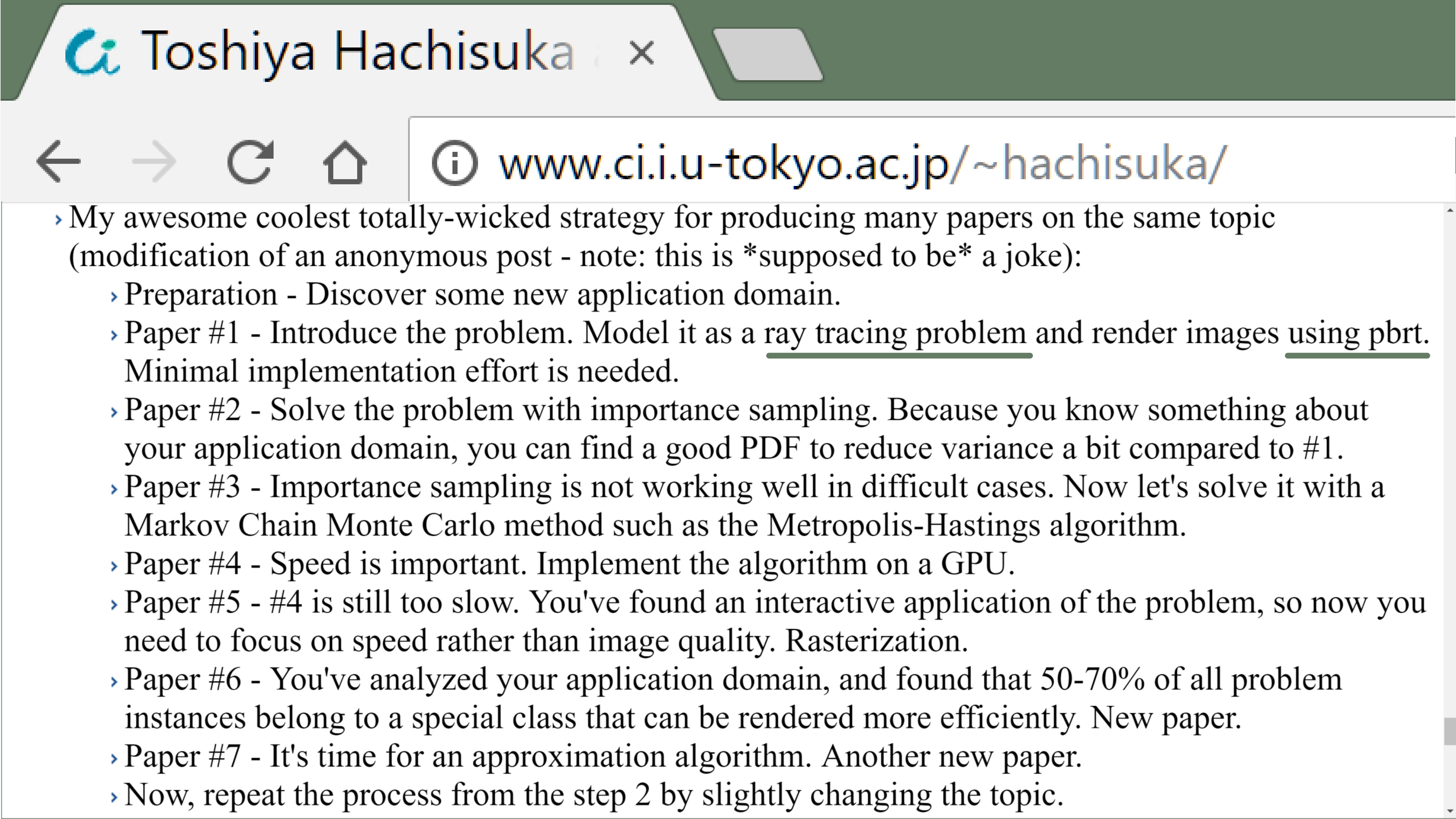


High-Performance Graphics 2017

Los Angeles | July 28–30, 2017

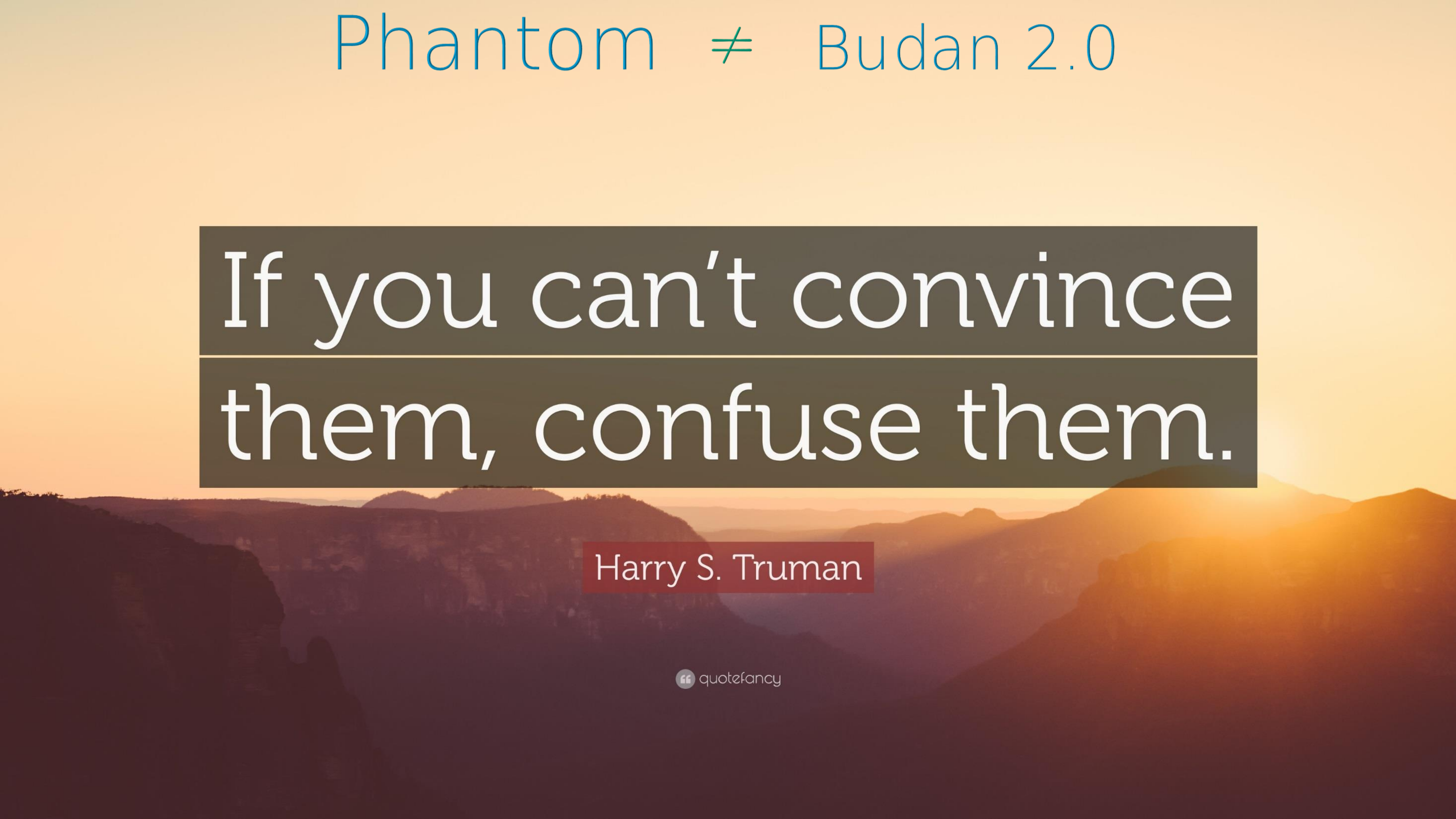


- My awesome coolest totally-wicked strategy for producing many papers on the same topic (modification of an anonymous post - note: this is *supposed to be* a joke):
  - Preparation - Discover some new application domain.
  - Paper #1 - Introduce the problem. Model it as a ray tracing problem and render images using pbrt. Minimal implementation effort is needed.
  - Paper #2 - Solve the problem with importance sampling. Because you know something about your application domain, you can find a good PDF to reduce variance a bit compared to #1.
  - Paper #3 - Importance sampling is not working well in difficult cases. Now let's solve it with a Markov Chain Monte Carlo method such as the Metropolis-Hastings algorithm.
  - Paper #4 - Speed is important. Implement the algorithm on a GPU.
  - Paper #5 - #4 is still too slow. You've found an interactive application of the problem, so now you need to focus on speed rather than image quality. Rasterization.
  - Paper #6 - You've analyzed your application domain, and found that 50-70% of all problem instances belong to a special class that can be rendered more efficiently. New paper.
  - Paper #7 - It's time for an approximation algorithm. Another new paper.
  - Now, repeat the process from the step 2 by slightly changing the topic.



- My awesome coolest totally-wicked strategy for producing many papers on the same topic (modification of an anonymous post - note: this is *supposed to be* a joke):
  - Preparation - Discover some new application domain.
  - Paper #1 - Introduce the problem. Model it as a ray tracing problem and render images using pbrt. Minimal implementation effort is needed.
  - Paper #2 - Solve the problem with importance sampling. Because you know something about your application domain, you can find a good PDF to reduce variance a bit compared to #1.
  - Paper #3 - Importance sampling is not working well in difficult cases. Now let's solve it with a Markov Chain Monte Carlo method such as the Metropolis-Hastings algorithm.
  - Paper #4 - Speed is important. Implement the algorithm on a GPU.
  - Paper #5 - #4 is still too slow. You've found an interactive application of the problem, so now you need to focus on speed rather than image quality. Rasterization.
  - Paper #6 - You've analyzed your application domain, and found that 50-70% of all problem instances belong to a special class that can be rendered more efficiently. New paper.
  - Paper #7 - It's time for an approximation algorithm. Another new paper.
  - Now, repeat the process from the step 2 by slightly changing the topic.

Phantom $\neq$ Budan 2.0

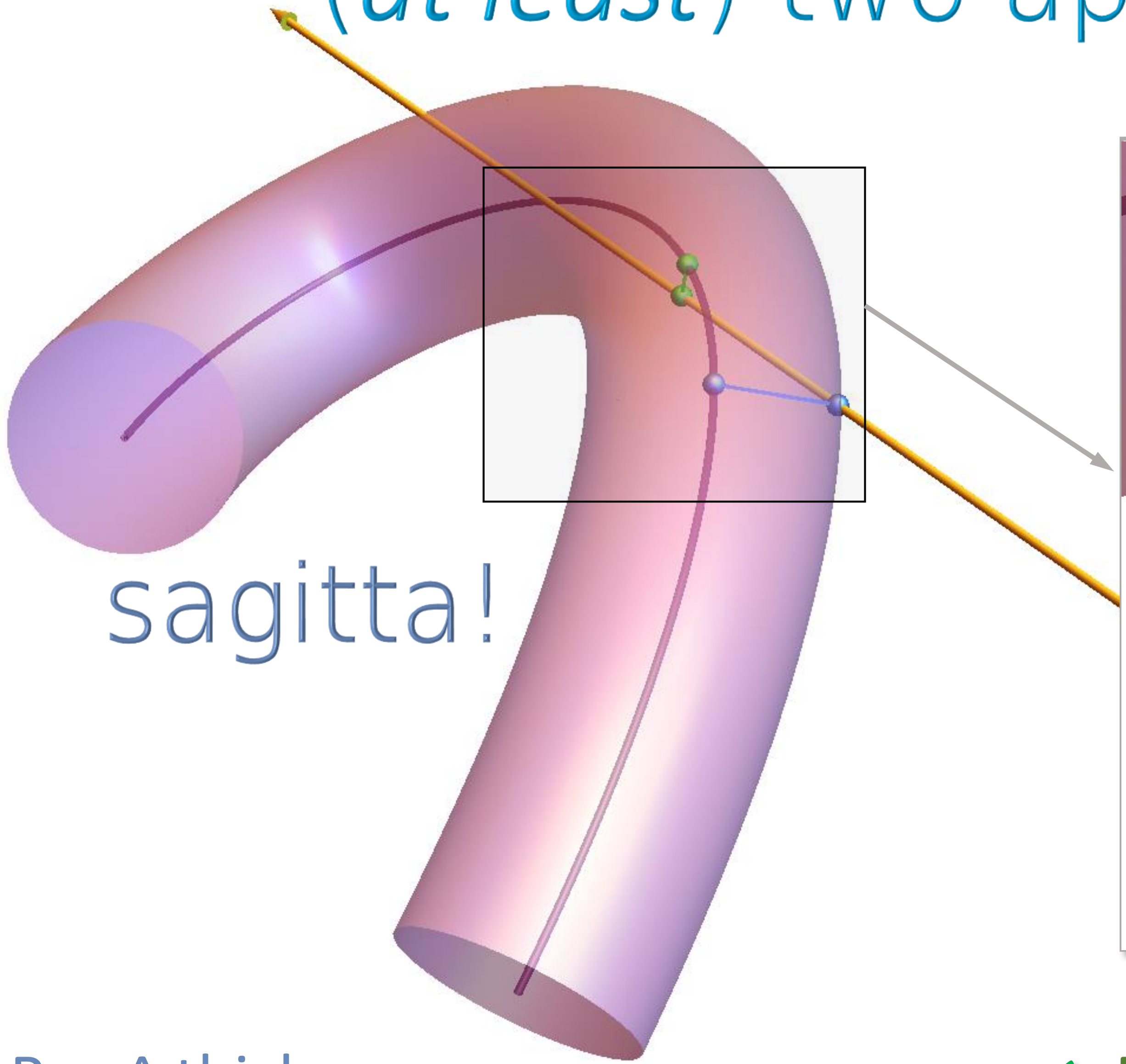


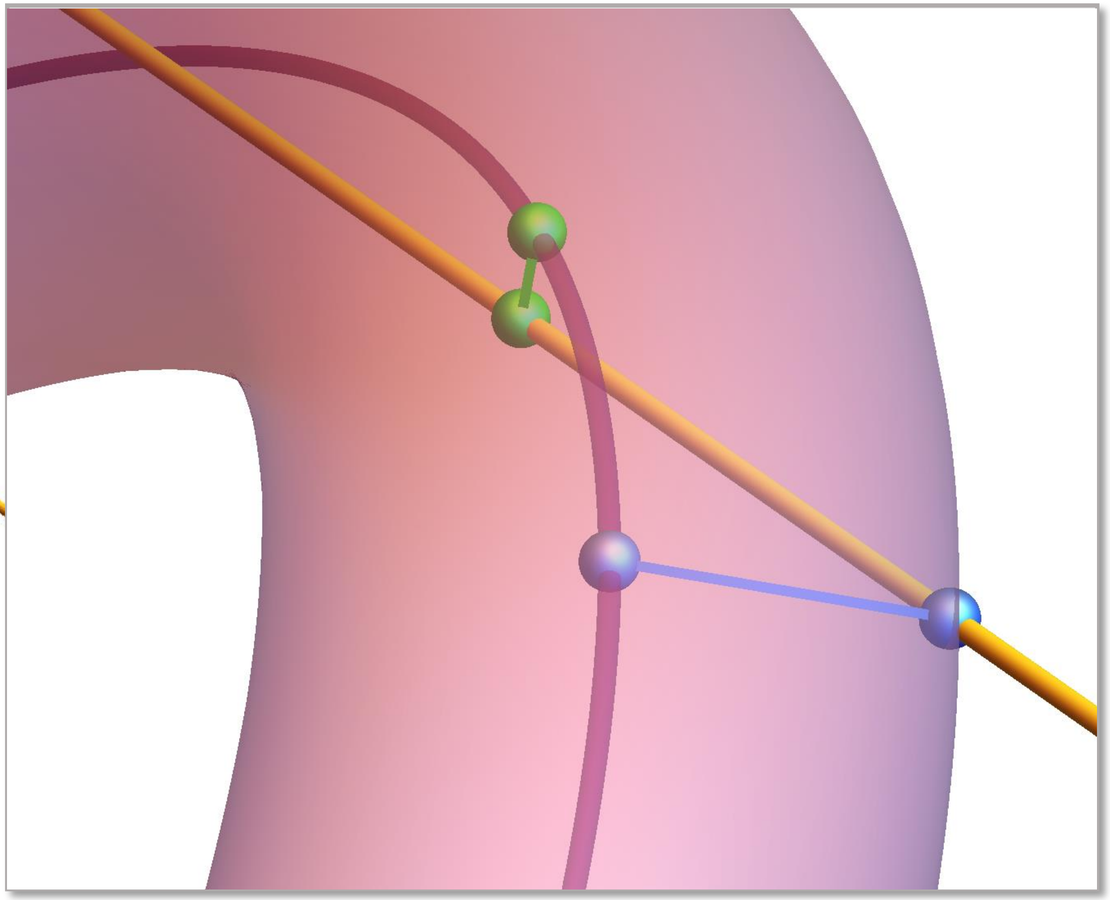
If you can't convince them, confuse them.

Harry S. Truman

quotefancy

(*at least*) two approaches

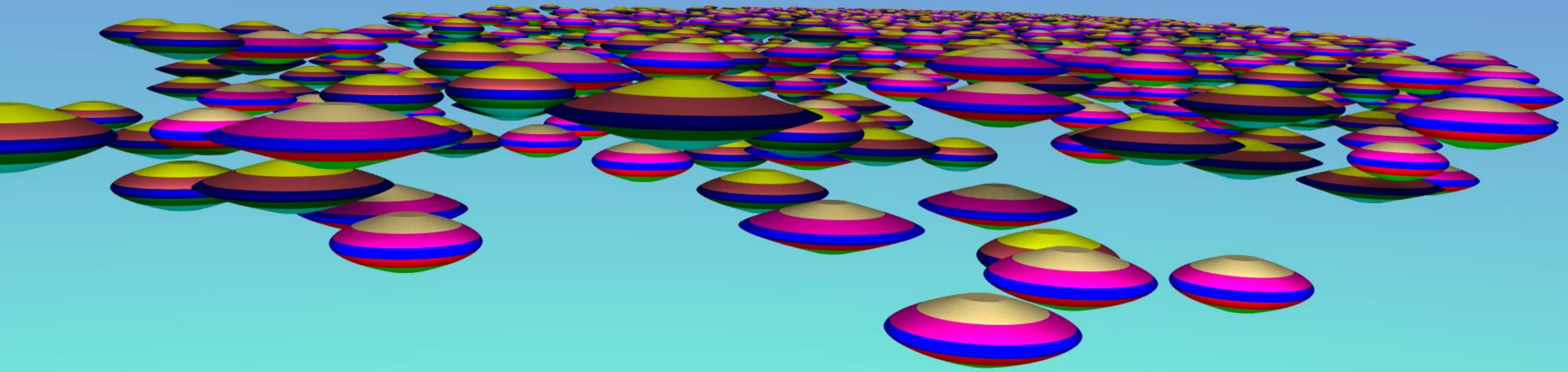




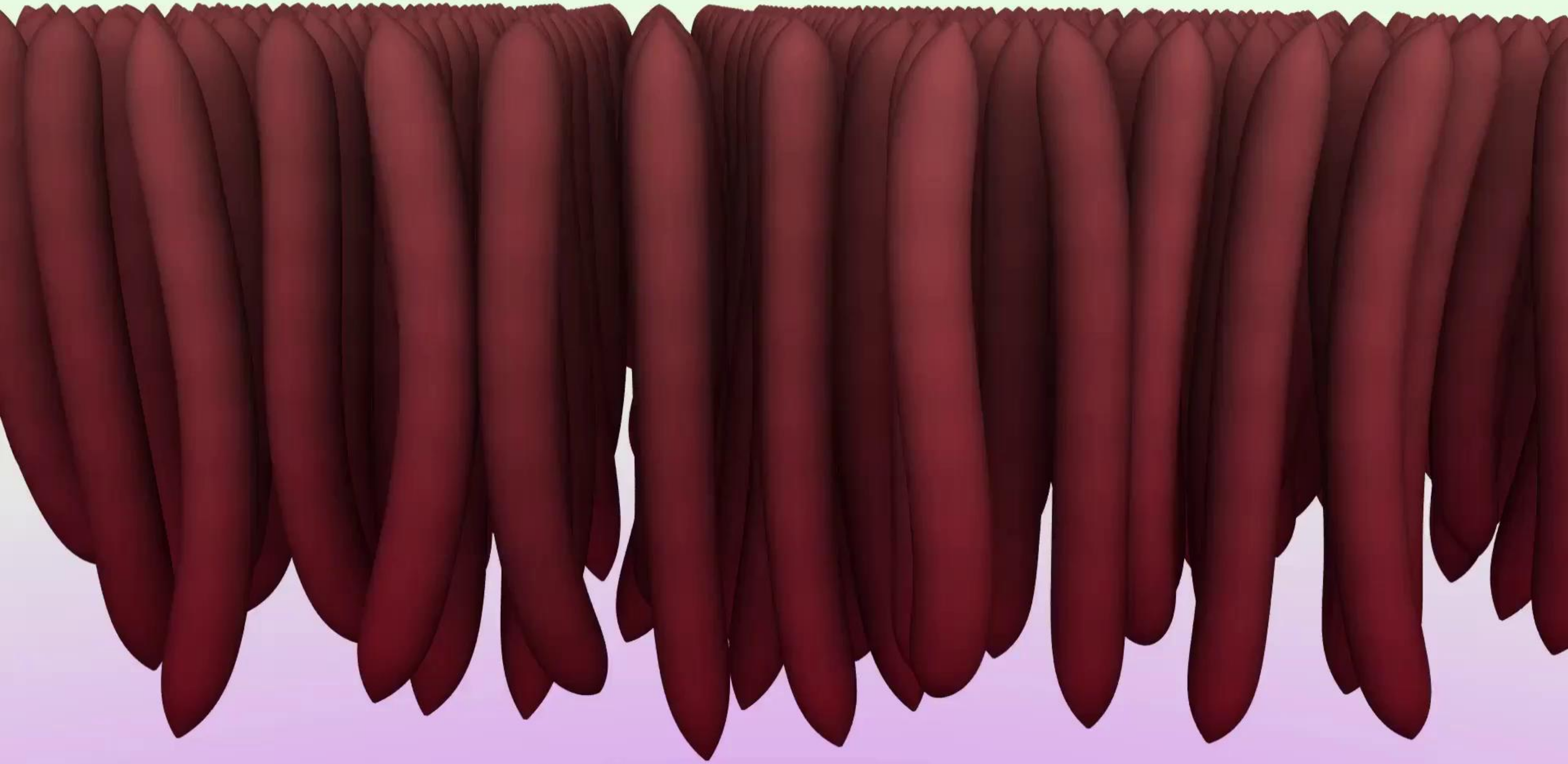
- <u>Ray ^ thick curve</u>
  more thrilling

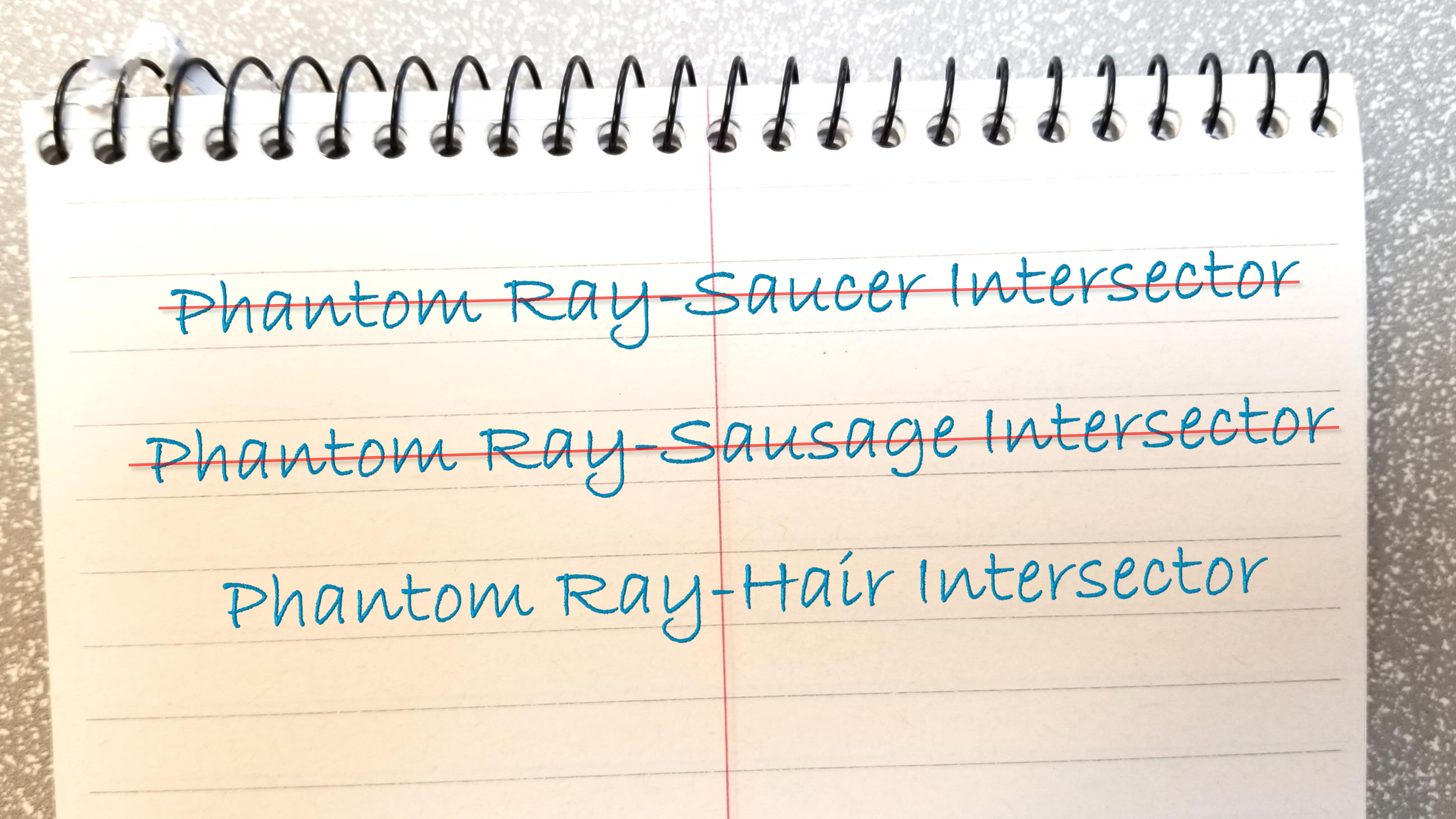
✓ <u>Minimum distance</u>
more universal (and easier)

Invasion of Flying Saucers



Sausage Factory





~~Phantom Ray-Saucer Intersector~~

~~Phantom Ray-Sausage Intersector~~

Phantom Ray-Hair Intersector

The ‘phantom’ name according to the reviewers

- the title is a bit strange and confusing
- hard to follow the real "story“
- a bit confusing
- 'phathom intersector' is kind of confusing
- totally disgusting brain f.rt

The ‘phantom’ name according to the reviewers

- the title is a bit strange and confusing
- hard to follow the real "story“
- a bit confusing
- 'phathom intersector' is kind of confusing
- totally disgusting brain f.rt

Phathom | Hardline Ent

Secure | https://hardlineent.com/artists/phathom/

Home News Artists Releases Videos Contact Store

Phathom

PHATHOM's, (originally called Section 8), four members (Anton, Jason, Matt and Yannick)

Members

Matthew Sikora - Vocals/Guitar
Yannick d' Assgnies - Bass
Anton Tilgren - Drums
Jason Zielonka - Guitar

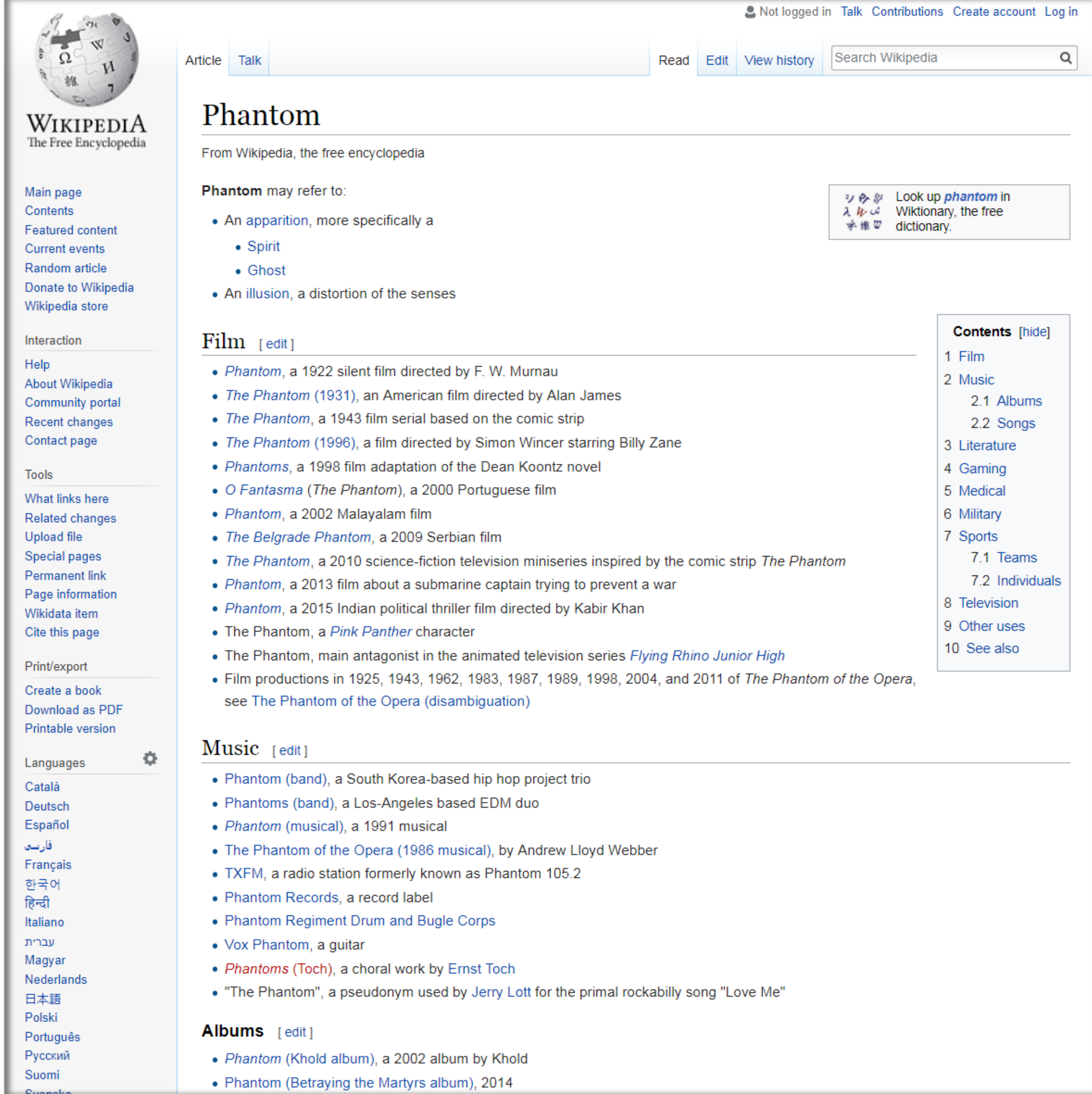
Phathom Links

- Facebook
- iTunes
- Last.FM
- MySpace
- PureVolume
- Reverb Nation
- Website

Phathom News

Twitter

Rich history of a ‘Phantom’ name



Not logged in Talk Contributions Create account Log in

Article Talk Read Edit View history Search Wikipedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
Català
Deutsch
Español
فارسی
Français
한국어
हिन्दी
Italiano
עברית
Magyar
Nederlands
日本語
Polski
Português
Русский
Suomi

Phantom

From Wikipedia, the free encyclopedia

Phantom may refer to:

Look up *phantom* in Wiktionary, the free dictionary.

- An apparition, more specifically a
  - Spirit
  - Ghost
- An illusion, a distortion of the senses

Film [edit]

- *Phantom*, a 1922 silent film directed by F. W. Murnau
- *The Phantom* (1931), an American film directed by Alan James
- *The Phantom*, a 1943 film serial based on the comic strip
- *The Phantom* (1996), a film directed by Simon Wincer starring Billy Zane
- *Phantoms*, a 1998 film adaptation of the Dean Koontz novel
- *O Fantasma* (*The Phantom*), a 2000 Portuguese film
- *Phantom*, a 2002 Malayalam film
- *The Belgrade Phantom*, a 2009 Serbian film
- *The Phantom*, a 2010 science-fiction television miniseries inspired by the comic strip *The Phantom*
- *Phantom*, a 2013 film about a submarine captain trying to prevent a war
- *Phantom*, a 2015 Indian political thriller film directed by Kabir Khan
- The Phantom, a *Pink Panther* character
- The Phantom, main antagonist in the animated television series *Flying Rhino Junior High*
- Film productions in 1925, 1943, 1962, 1983, 1987, 1989, 1998, 2004, and 2011 of *The Phantom of the Opera*, see The Phantom of the Opera (disambiguation)

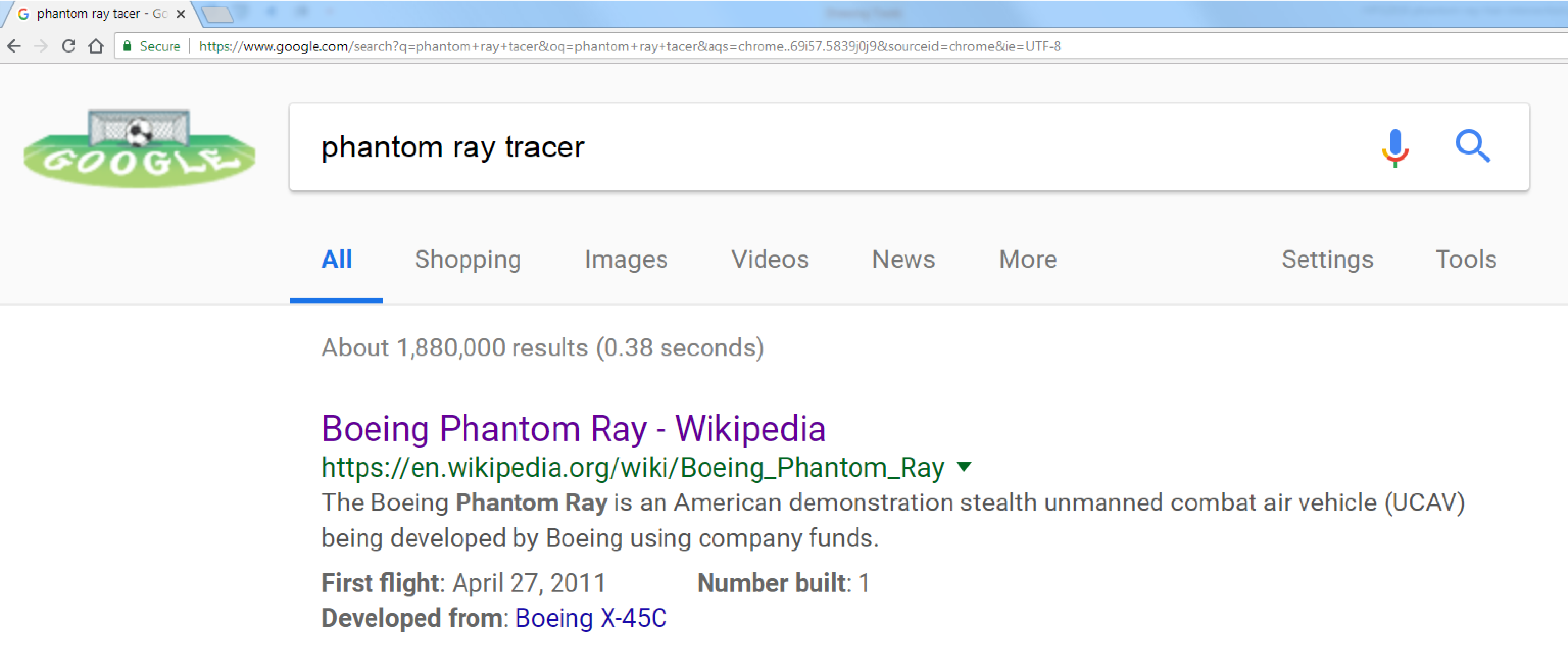
Music [edit]

- Phantom (band), a South Korea-based hip hop project trio
- Phantoms (band), a Los-Angeles based EDM duo
- *Phantom* (musical), a 1991 musical
- The Phantom of the Opera (1986 musical), by Andrew Lloyd Webber
- TXFM, a radio station formerly known as Phantom 105.2
- Phantom Records, a record label
- Phantom Regiment Drum and Bugle Corps
- Vox Phantom, a guitar
- *Phantoms* (Toch), a choral work by Ernst Toch
- "The Phantom", a pseudonym used by Jerry Lott for the primal rockabilly song "Love Me"

Albums [edit]

- *Phantom* (Khold album), a 2002 album by Khold
- Phantom (Betraying the Martyrs album), 2014

Jacco Bikker aka ‘Phantom’



phantom ray tacer - Go

Secure | https://www.google.com/search?q=phantom+ray+tacer&oq=phantom+ray+tacer&aqs=chrome..69i57.5839j0j9&sourceid=chrome&ie=UTF-8

phantom ray tracer

All Shopping Images Videos News More Settings Tools

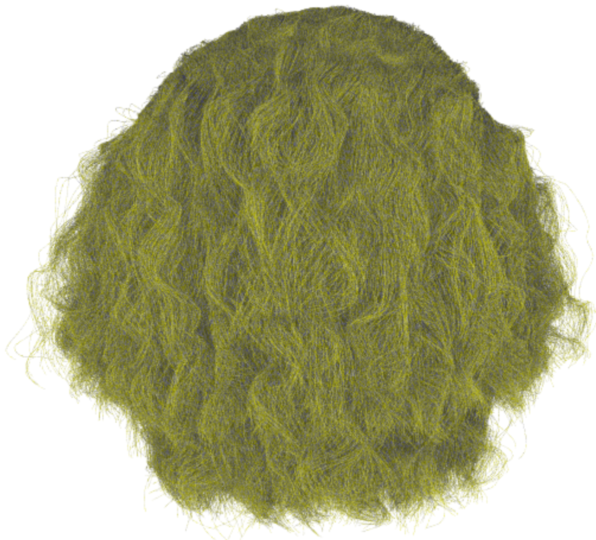
About 1,880,000 results (0.38 seconds)

Boeing Phantom Ray - Wikipedia
https://en.wikipedia.org/wiki/Boeing_Phantom_Ray
The Boeing **Phantom Ray** is an American demonstration stealth unmanned combat air vehicle (UCAV) being developed by Boeing using company funds.

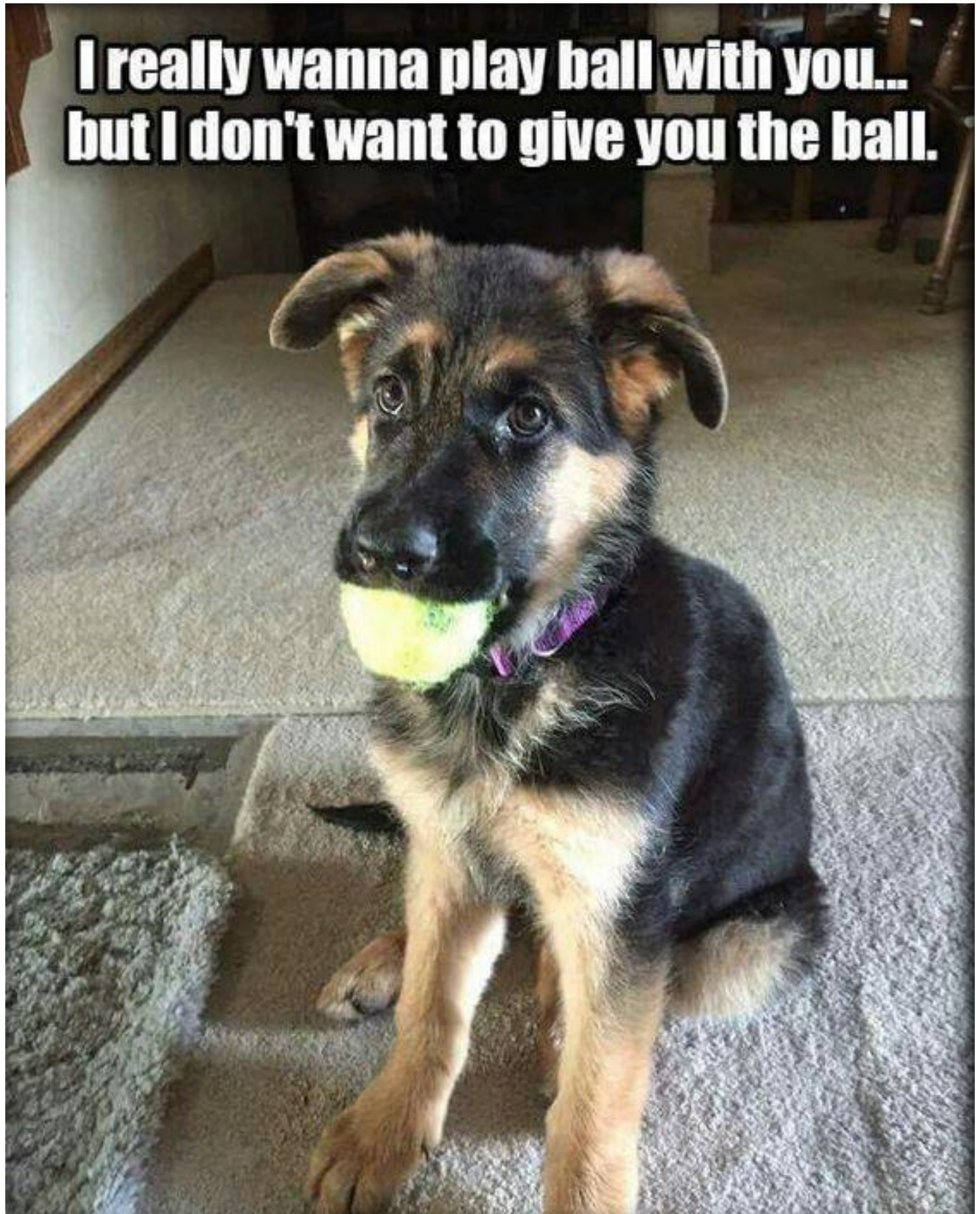
First flight: April 27, 2011 **Number built**: 1
Developed from: Boeing X-45C

Global Services Unlimited Support
BOEING
Phantom Ray
GE
WOODWARD
Honeywell

Let’s do an experiment…



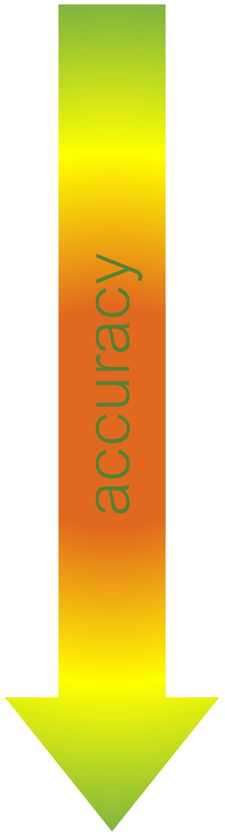
Now we have a dilemma…


I really wanna play ball with you...
but I don't want to give you the ball.

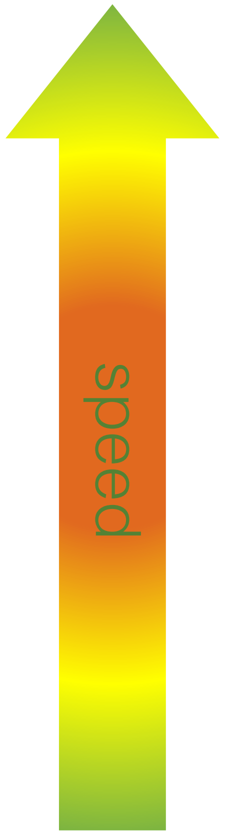
Now we have a dilemma…

- Any method works at intermediate distances
- (and something else has to be done @ far away)
- Accuracy matters at closeups
- If an ‘accurate’ intersector were significantly slower, we would have to resort to some complicated LOD scheme…

Taxonomy of ray-hair intersection methods



1. Texture approximations (2D and 3D)
   [Andersen et al. 2016; Hadap et al. 2007; Kajiya and Kay 1989; Lengyel et al. 2001; Petrovic et al. 2005; Ren et al. 2010; Sintorn and Assarsson 2009]
2. Fixed 3D cylinders
   [Sedaghat 2010; Martins 2016]
3. CPA (the closest point of approach) methods
   [Barringer et al. 2012; Chiang et al. 2015; Nakamaru and Ohno 2002; Qin et al. 2014; Reshetov 2017; Woop et al. 2014]
4. Accurate geometric methods
   [Wijk 1985; Bronsvoort and Klok 1985]



Fortunately…

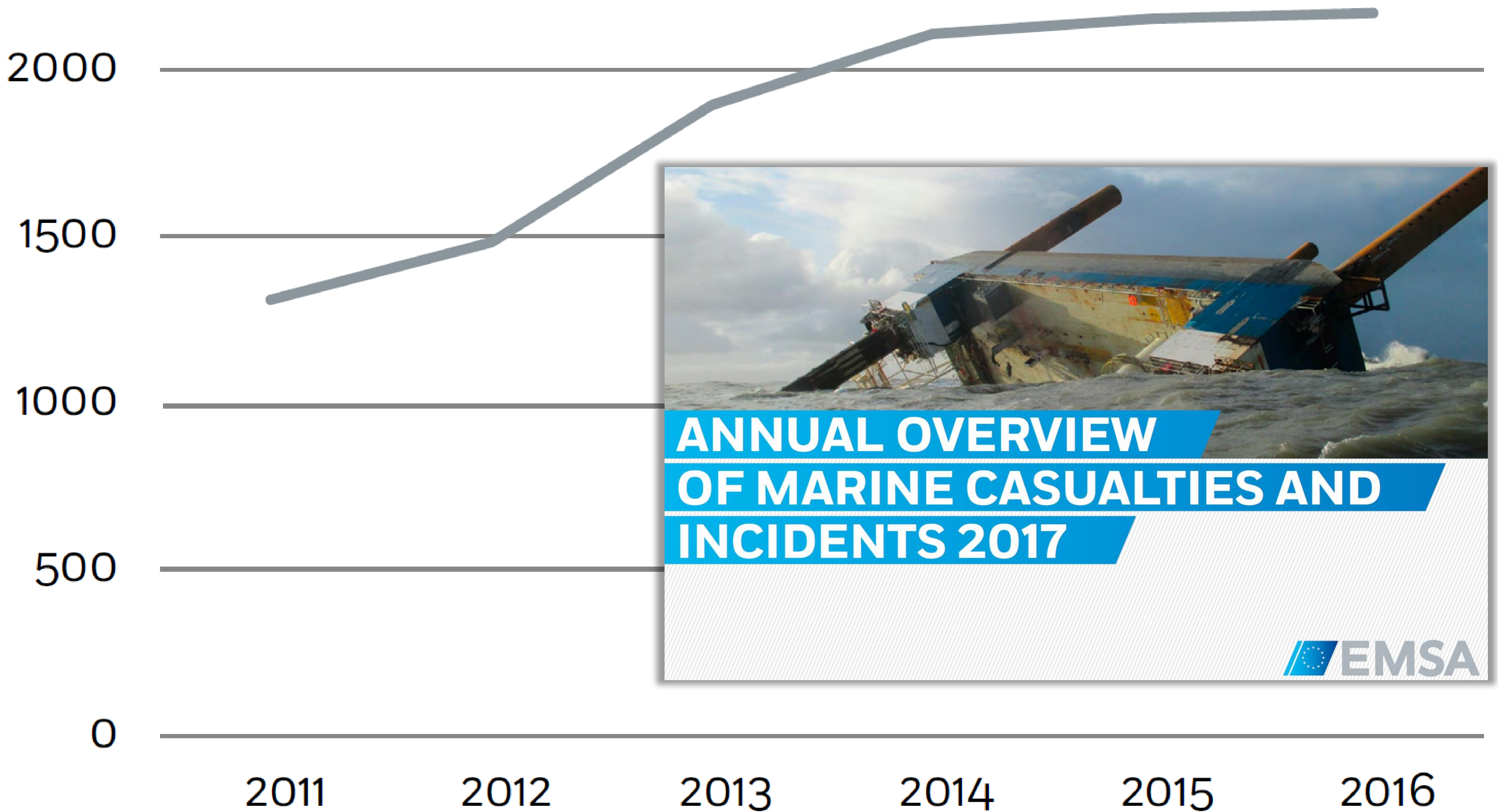
- Phantom intersector is 25% faster than the Budan one on CPU
- while Budan is 5X faster than the adaptive linearization methods
- and it is well suited for GPU implementation
- (I didn’t try implementing Budan on GPU)
- fixed cylinders still about 5% faster though by itself,
  but it is not worth any LOD complications…

Yay for CPA

- We’re talking only about CPA => bona fide ray/swept volume intersection
- CPA finds a minimum distance between a ray and a curve that might be useful in other applications (collisions)
- In fact, CPA was first proposed by Morell to reduce ship collisions

J. S. Morrel. 1961. The Physics of Collision at Sea. *Journal of Navigation* 14, 2 (1961), 163–184.

Casualty with a ship



ANNUAL OVERVIEW
OF MARINE CASUALTIES AND
INCIDENTS 2017

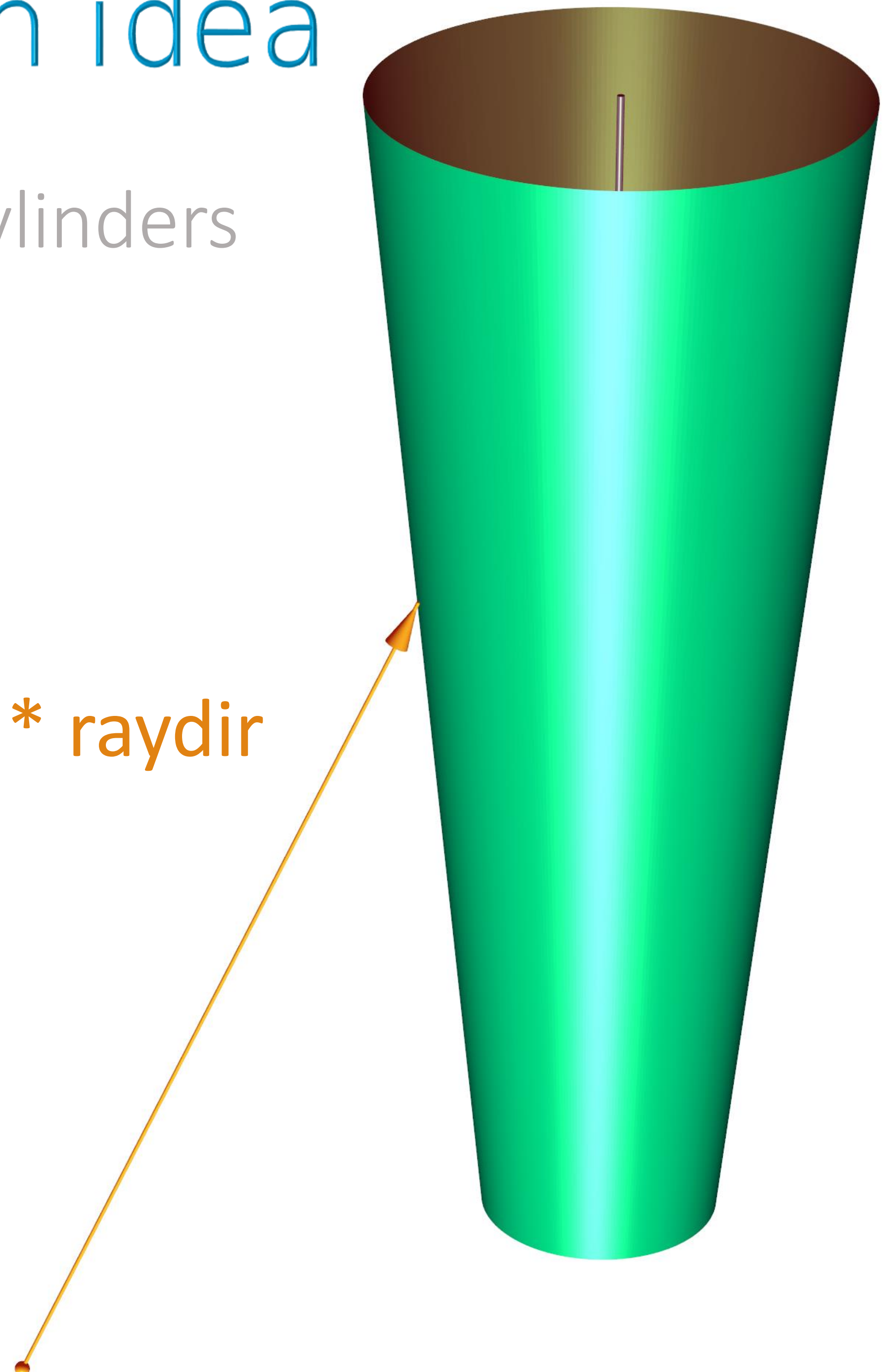
EMSA

Phantom xsector: the main idea

- Instead of approximating geometry with cylinders (or cones),
- we’ll use ray-cone intersection inside an iterative scheme.

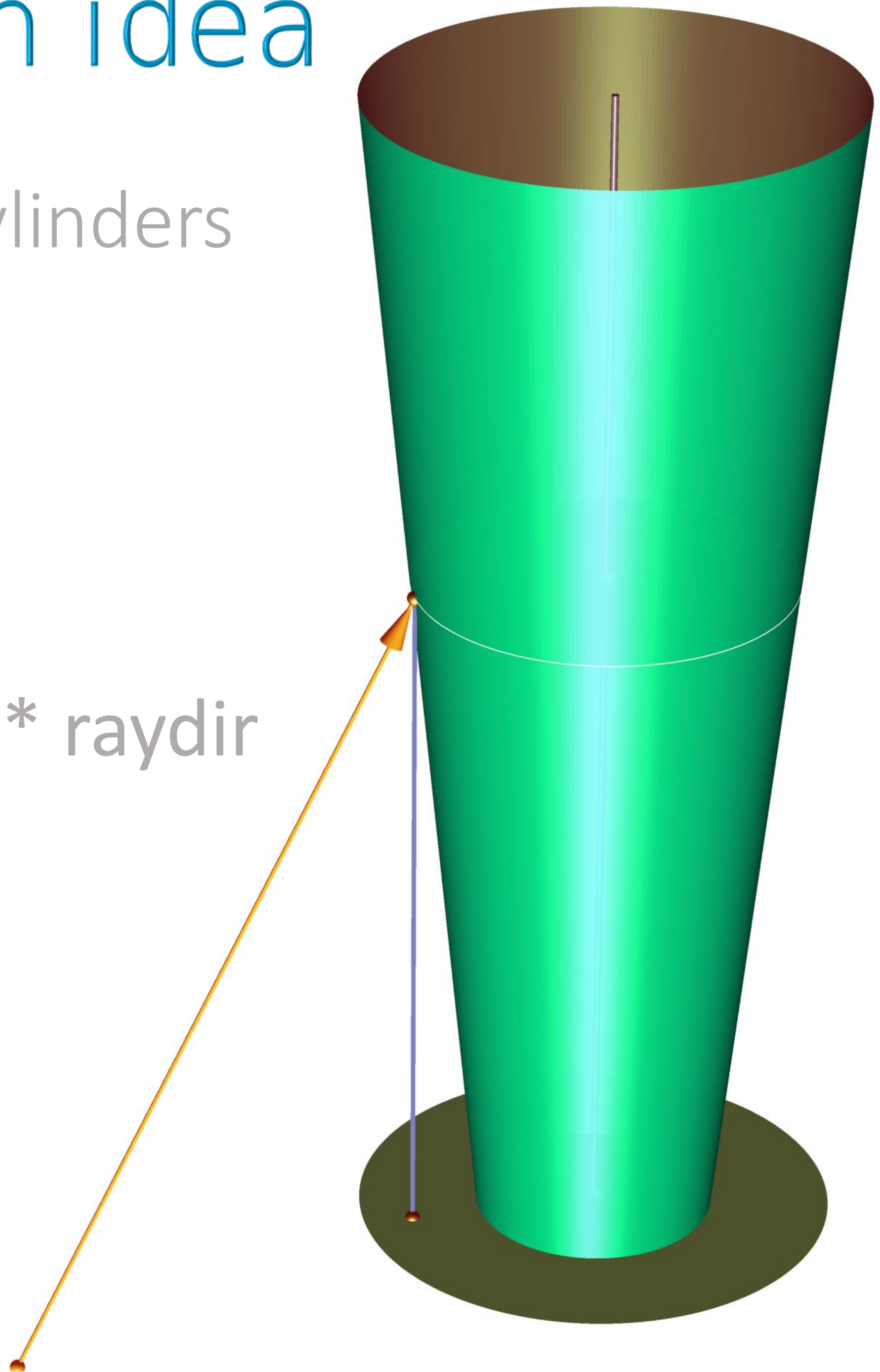
Phantom xsector: the main idea

- Instead of approximating geometry with cylinders (or cones),
- we'll use ray-cone intersection inside an iterative scheme.
- If the hair strand is a cone, we solve a quadratic equation to find s in rayorg + s * raydir



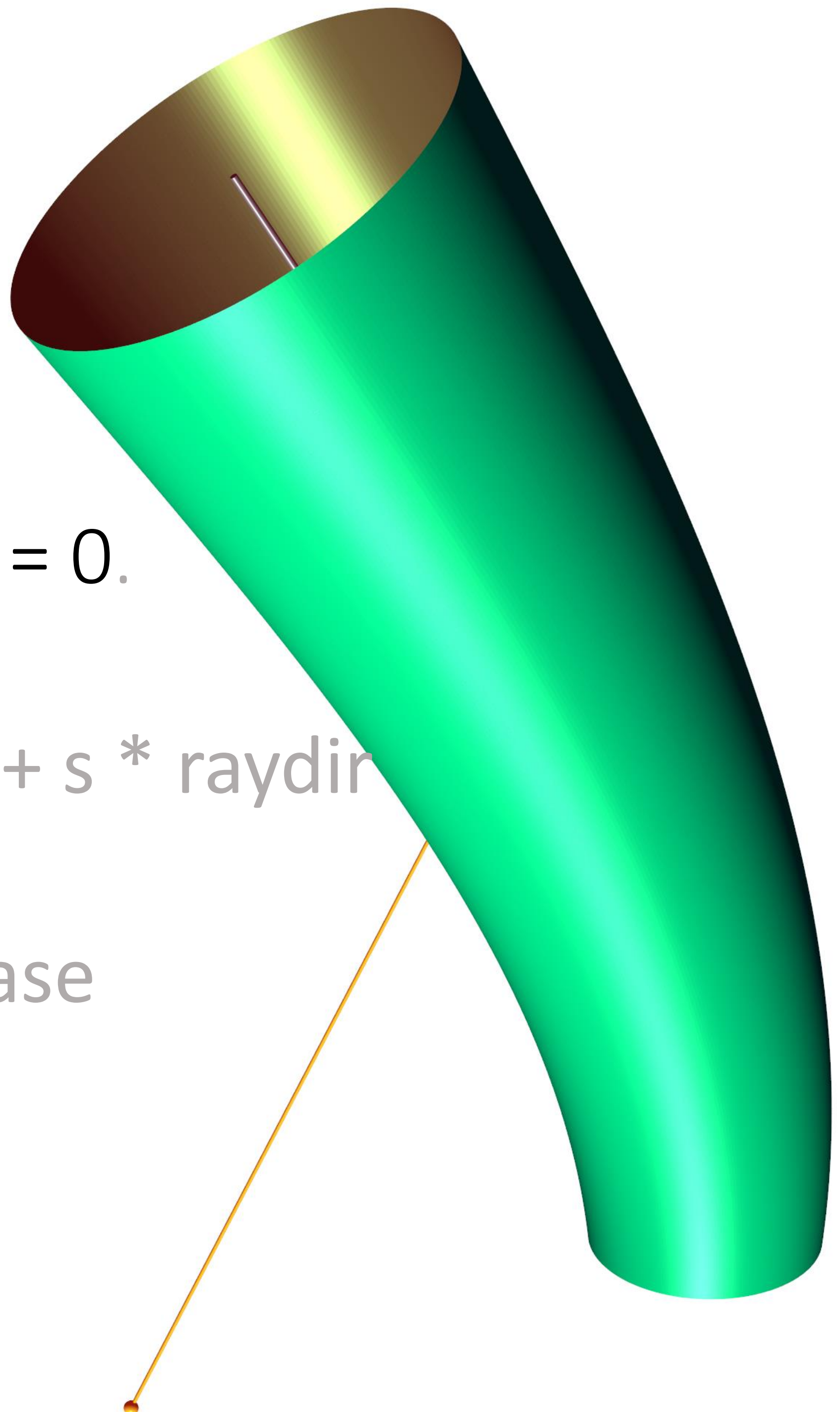
Phantom xsector: the main idea

- Instead of approximating geometry with cylinders (or cones),
- we'll use ray-cone intersection inside an iterative scheme.
- If the hair strand is a cone, we solve a quadratic equation to find s in **rayorg + s * raydir**
- To find the surface normal, we will need a distance to the cone's base



General case

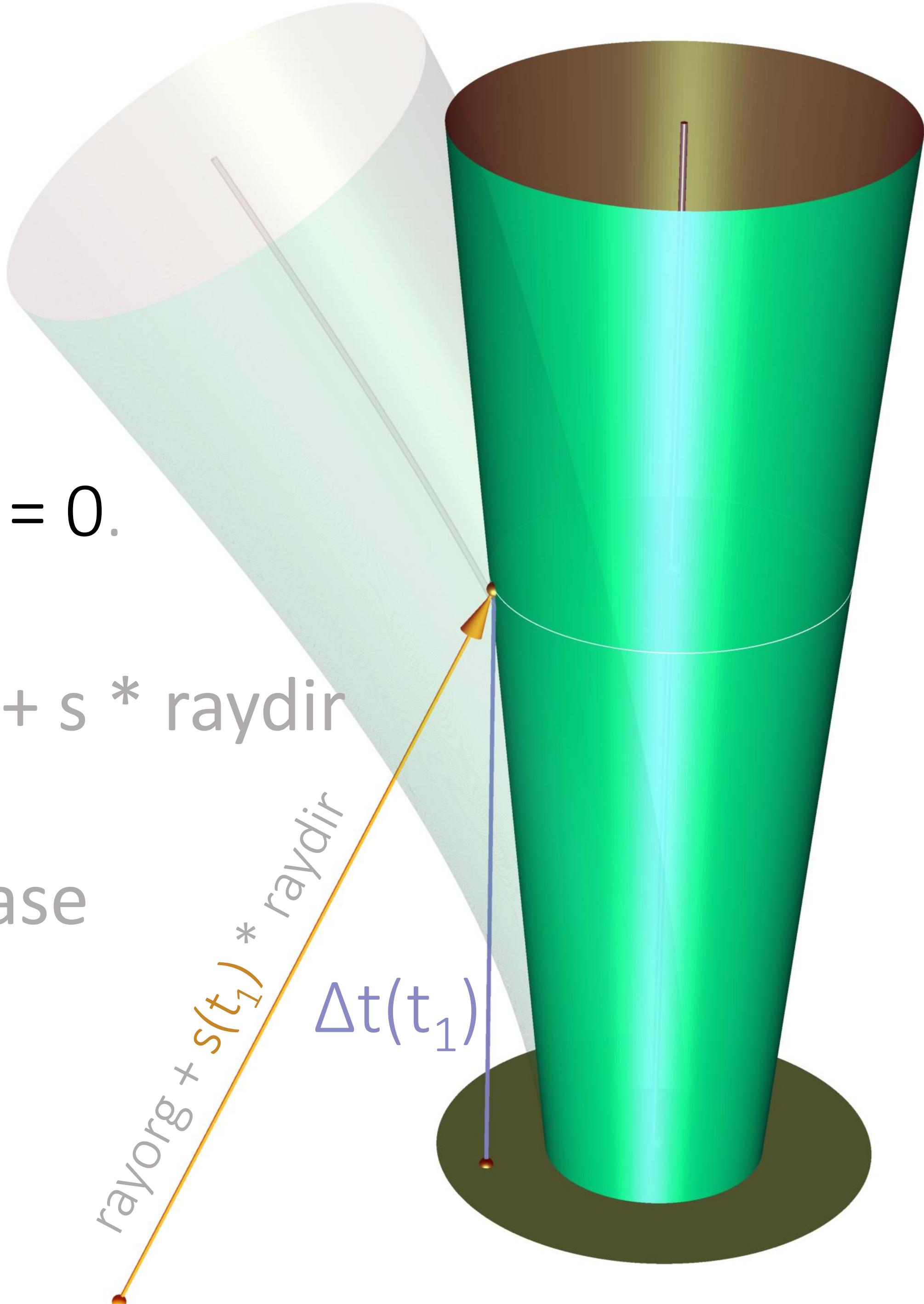
- Instead of approximating geometry,
- we'll use ray-cone intersection inside an **iterative scheme starting at $t_1 = 0$**.
- If the hair strand is a cone, we solve a quadratic equation to find s in **rayorg + s * raydir**
- To find the surface normal, we will need a **distance to the cone's base**



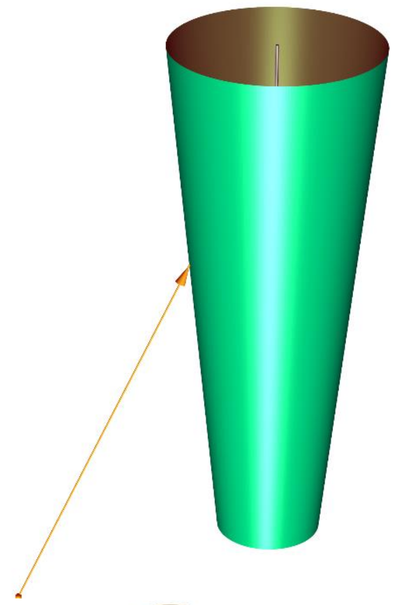
General case

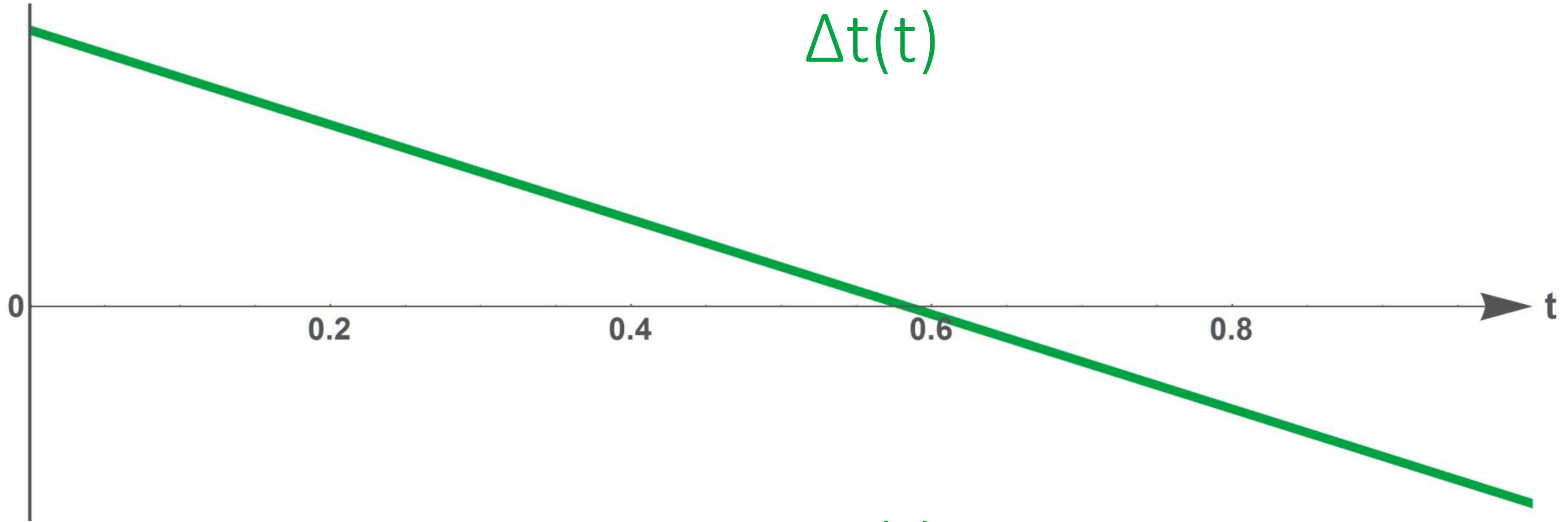
- Instead of approximating geometry,
- we'll use ray-cone intersection inside an **iterative scheme starting at $t_1 = 0$**.
- If the hair strand is a cone, we solve a quadratic equation to find s in **rayorg + s * raydir**
- To find the surface normal, we will need a **distance to the cone's base**

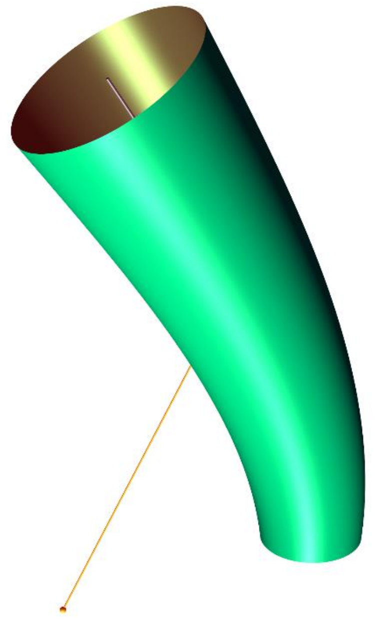
1. find cone(t_1) $\rightarrow$ s(t_1) $\rightarrow$ $\Delta t(t_1)$
2. then set $t_2 = t_1 + \Delta t(t_1)$

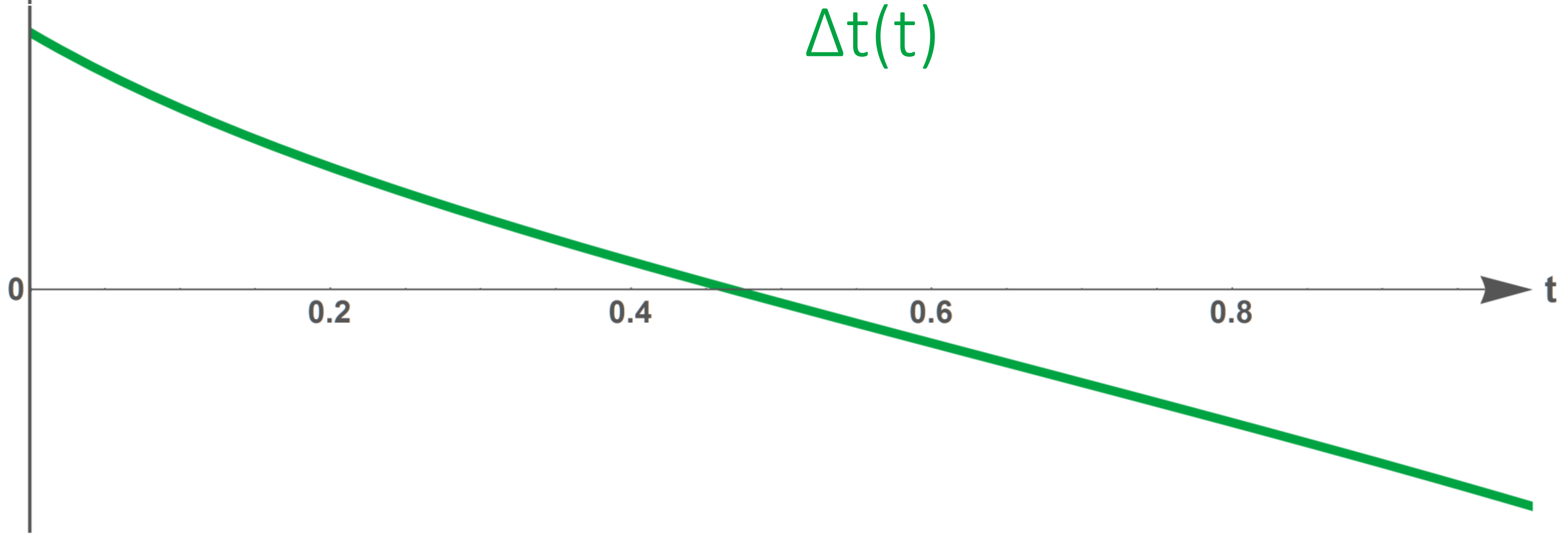


It works (mostly) OK









There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)

2. $\Delta t(t)$ function could have ∞ values (ray || cone)

There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)
   i.e. determinant of a quadratic equation < 0

2. $\Delta t(t)$ function could have ∞ values (ray || cone)

There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)
   i.e. determinant of a quadratic equation < 0
   in such a case we just set it to 0

2. $\Delta t(t)$ function could have ∞ values (ray || cone)

There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)
   i.e. determinant of a quadratic equation < 0
   in such a case we just set it to 0
   ≡ the intersection with the padded cone

2. Δt(t) function could have ∞ values (ray || cone)

There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)
   i.e. determinant of a quadratic equation < 0
   in such a case we just set it to 0
   $\equiv$ the intersection with the padded cone
   $\models$ 'phantom' name
2. $\Delta t(t)$ function could have ∞ values (ray || cone)

There are two (m or) compli ions

coul ne(t)

air)

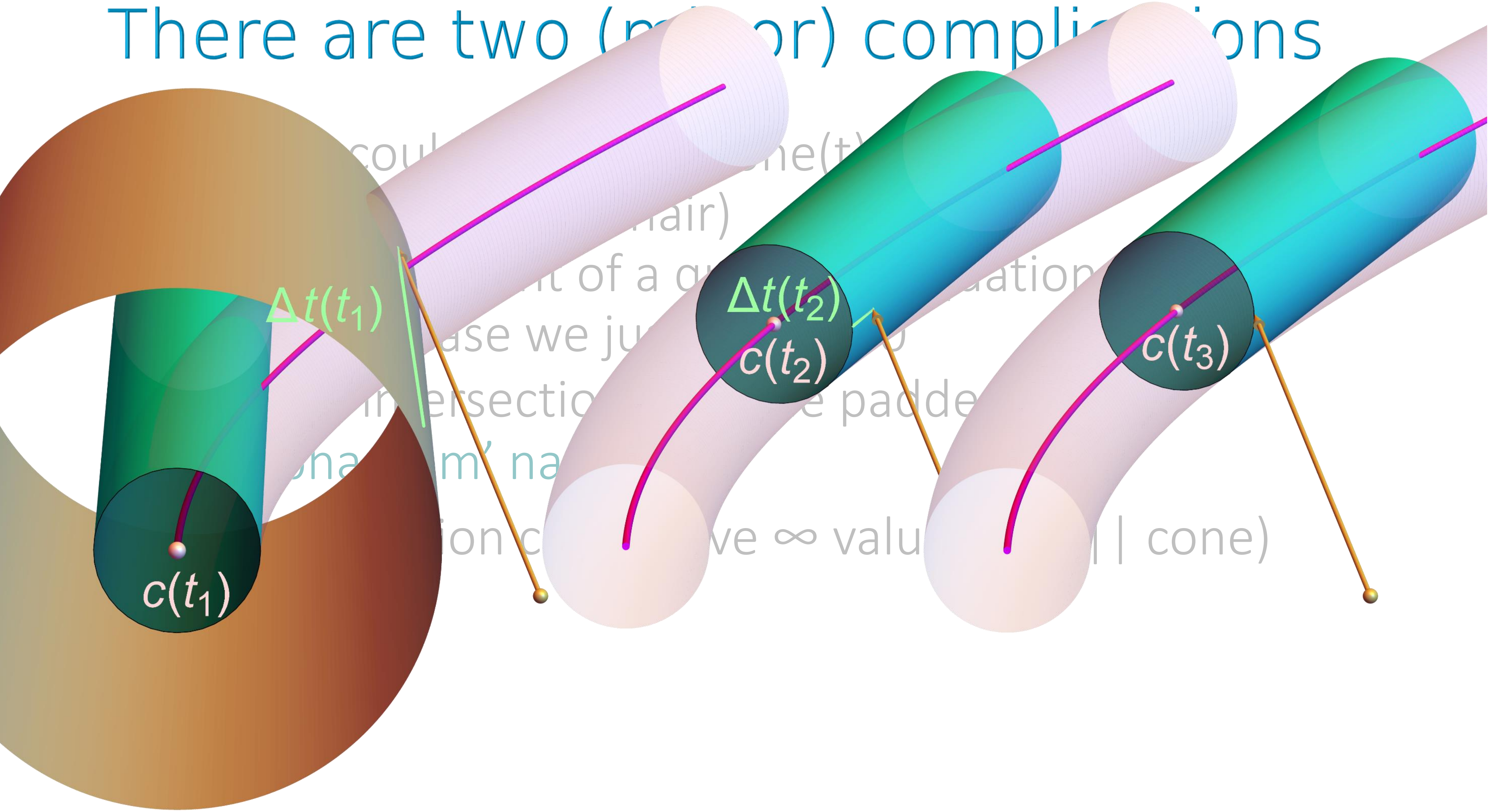
t of a q uation

ase we ju o

in ersectio e padde

ha m' na

ion c ve ∞ valu || cone)

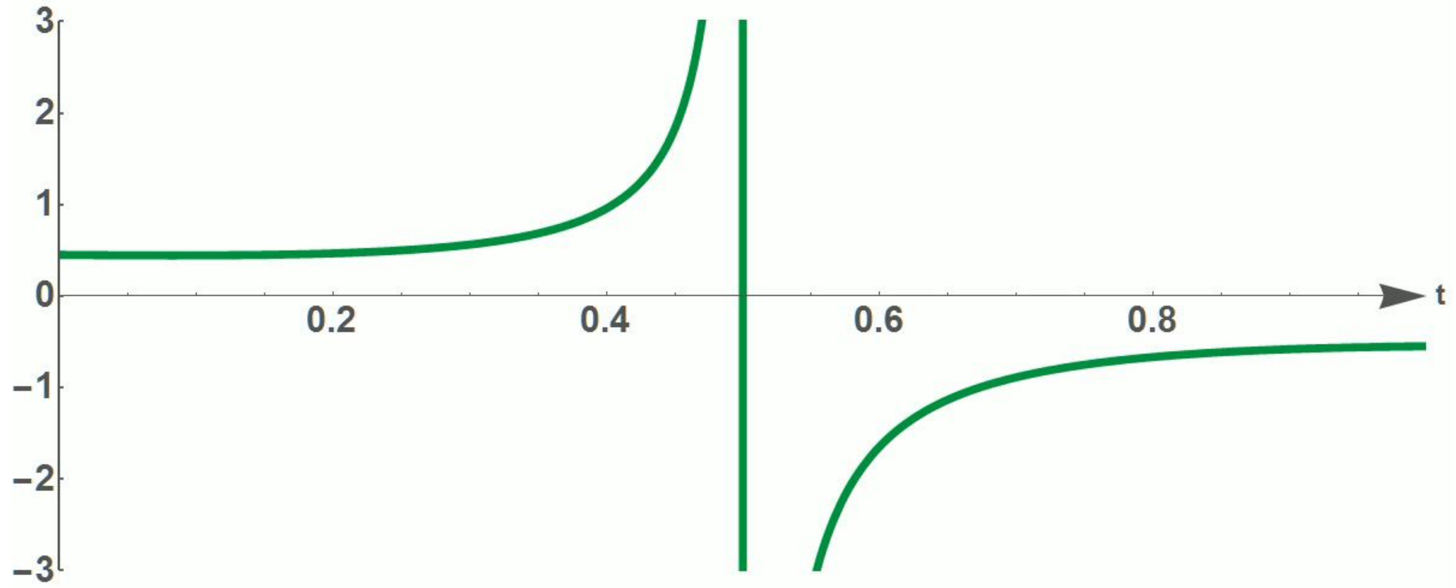


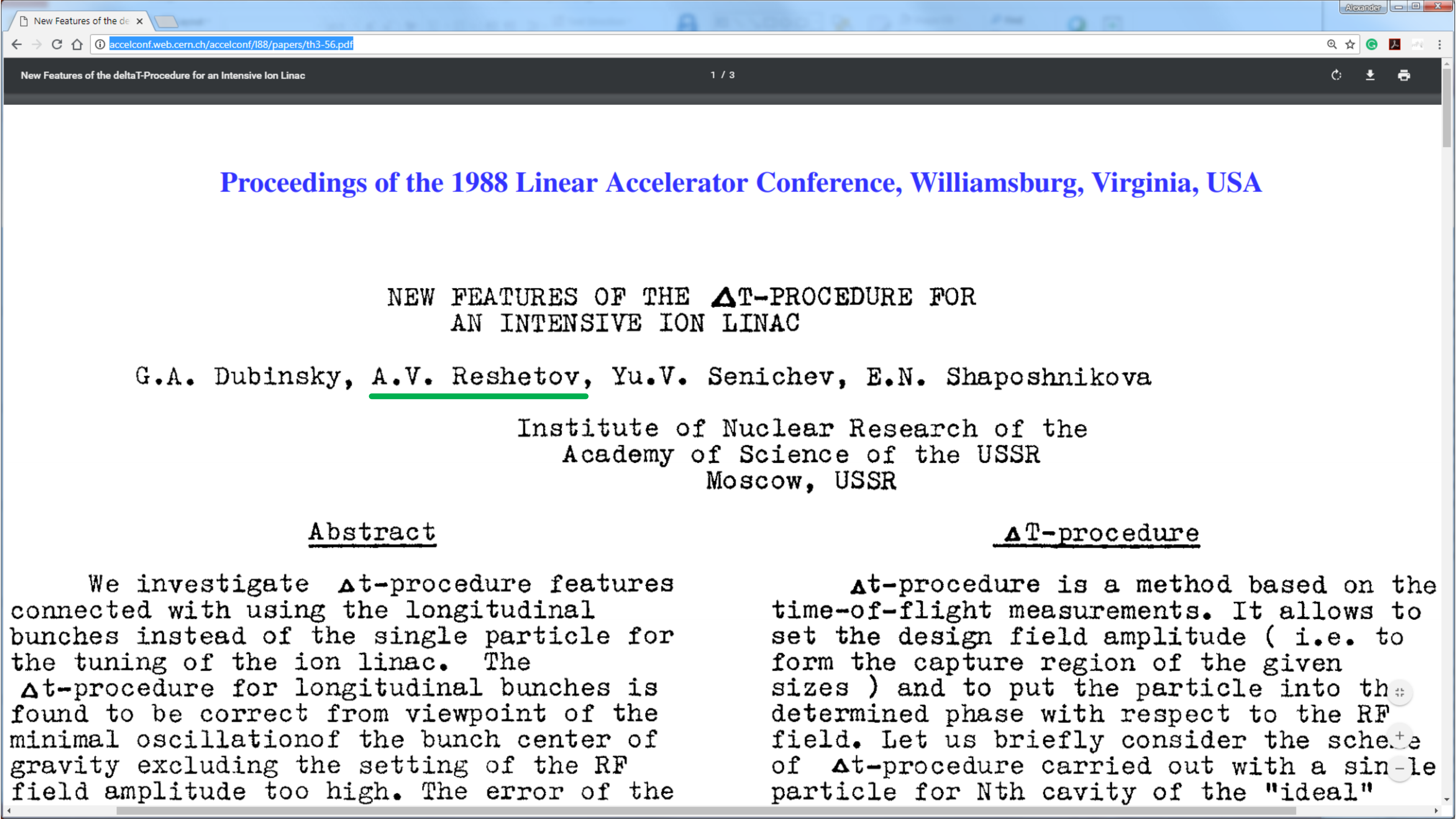
There are two (minor) complications

1. There could be no ray/cone(t) intersections (especially for thin hair)
   i.e. determinant of a quadratic equation < 0
   in such a case we just set it to 0
   ≡ the intersection with the padded cone
   ⊨ ‘phantom’ name
2. Δt(t) function could have ∞ values (ray || cone)
   we need some kind of procedure to address this issue

$\Delta t(0.5) = \infty$

single root
two roots




New Features of the de
accelconf.web.cern.ch/accelconf/l88/papers/th3-56.pdf
New Features of the deltaT-Procedure for an Intensive Ion Linac
1 / 3
Alexander

Proceedings of the 1988 Linear Accelerator Conference, Williamsburg, Virginia, USA

NEW FEATURES OF THE ΔT-PROCEDURE FOR AN INTENSIVE ION LINAC

G.A. Dubinsky, A.V. Reshetov, Yu.V. Senichev, E.N. Shaposhnikova

Institute of Nuclear Research of the
Academy of Science of the USSR
Moscow, USSR

Abstract

We investigate Δt-procedure features connected with using the longitudinal bunches instead of the single particle for the tuning of the ion linac. The Δt-procedure for longitudinal bunches is found to be correct from viewpoint of the minimal oscillationof the bunch center of gravity excluding the setting of the RF field amplitude too high. The error of the

ΔT-procedure

Δt-procedure is a method based on the time-of-flight measurements. It allows to set the design field amplitude (i.e. to form the capture region of the given sizes) and to put the particle into th[illegible] determined phase with respect to the RF field. Let us briefly consider the sche[illegible]e of Δt-procedure carried out with a sin[illegible]le particle for Nth cavity of the "ideal"

- Most commonly, Newton - Raphson method is used for root finding…

- Most commonly, Newton's method is used for root finding…
- He also invented cat doors

Newton's experiments were interrupted constantly by his cats scratching at his office door, so he summoned the Cambridge carpenter and had him saw two holes in his door: a large hole for the mother cat and a small one for her kittens.

https://science.howstuffworks.com/innovation/famous-inventors/5-isaac-newton-inventions2.htm

Why we will not use Newton's method:

$D[dt[t, \{\{c_{0x}, c_{1x}, c_{2x}, c_{3x}\}, \{c_{0y}, c_{1y}, c_{2y}, c_{3y}\}, \{c_{0z}, c_{1z}, c_{2z}, c_{3z}\}\}, r[t], r'[t]], t]$

$\Delta t[t] =$

```
inline bool intersect(float r, float dr) {
  // cone is defined by base center c0, radius r,
  // axis cd, and slant dr
  float r2  = r * r;  // dr could be either positive
  float drr = r * dr; // or negative (0 for cylinder)

  float ddd = cd.x*cd.x + cd.y*cd.y; // all possible
  dp        = c0.x*c0.x + c0.y*c0.y; // combinations
  float cdd = c0.x*cd.x + c0.y*cd.y; // of x*y terms
  float cxd = c0.x*cd.y − c0.y*cd.x; // (c0 × cd)z

  float c = ddd;                     // compute a,b,c in
  float b = cd.z * (drr − cdd);      //    a − 2 b s + c s²
  float cdz2 = cd.z*cd.z;            //    (s for ray ∩ cone)
  ddd += cdz2;                       // now it is cd·cd
  float a = 2*drr*cdd + cxd*cxd − ddd*r2 + dp*cdz2;
 #if defined(KEEP_DR2)               // dr² adjustments
  float qs = (dr*dr)/ddd;            // (it does not help
  a −= qs * cdd*cdd;                 //  much with neither
  b −= qs * cd.z*cdd;                //  performance nor
  c −= qs * cdz2;                    //  accuracy)
 #endif
  // We will add c0.z to s and sp latter if needed
  float det = b*b − a*c;      // for a − 2 b s + c s²
  s   = (b − (det > 0? sqrt(det) : 0))/c; // c > 0
  dt  = (s*cd.z − cdd)/ddd;   // wrt t
  dc  = s*s + dp;             // |(ray ∩ cone)  − c0|²
  sp  = cdd/cd.z;             // will add c0.z latter
  dp += sp*sp;                // |(ray ∩ plane) − c0|²

  return det > 0;             // true (real) or false (phantom)
}
```

$\Delta t'[t] =$ [illegible]

Why we will not use Newton's method:

$D[dt[t, \{\{c_{0x},c_{1x},c_{2x},c_{3x}\},\{c_{0y},c_{1y},c_{2y},c_{3y}\},\{c_{0z},c_{1z},c_{2z},c_{3z}\}\}, r[t],r'[t]], t]$

$\Delta t[t] =$

```
inline bool intersect(float r, float dr) {
  // cone is defined by base center c0, radius r,
  // axis cd, and slant dr
  float r2  = r * r;  // dr could be either positive
  float drr = r * dr; // or negative (0 for cylinder)

  float ddd = cd.x*cd.x + cd.y*cd.y; // all possible
  dp        = c0.x*c0.x + c0.y*c0.y; // combinations
  float cdd = c0.x*cd.x + c0.y*cd.y; // of x*y terms
  float cxd = c0.x*cd.y - c0.y*cd.x; // (c0 × cd)z

  float c = ddd;                   // compute a,b,c in
  float b = cd.z * (drr - cdd);    //   a - 2 b s + c s^2
  float cdz2 = cd.z*cd.z;          //   (s for ray ∩ cone)
  ddd += cdz2;                     // now it is cd·cd
  float a = 2*drr*cdd + cxd*cxd - ddd*r2 + dp*cdz2;
 #if defined(KEEP_DR2)             // dr^2 adjustments
  float qs = (dr*dr)/ddd;          // (it does not help
  a -= qs * cdd*cdd;               //  much with neither
  b -= qs * cd.z*cdd;              //  performance nor
  c -= qs * cdz2;                  //  accuracy)
 #endif
  // We will add c0.z to s and sp latter if needed
  float det = b*b - a*c;      // for a - 2 b s + c s^2
  s   = (b - (det > 0? sqrt(det) : 0))/c; // c > 0
  dt  = (s*cd.z - cdd)/ddd;   // wrt t
  dc  = s*s + dp;             // |(ray ∩ cone)  - c0|^2
  sp  = cdd/cd.z;             // will add c0.z latter
  dp += sp*sp;                // |(ray ∩ plane) - c0|^2

  return det > 0;             // true (real) or false (phantom)
}
```

$\Delta t'[t] = -$ [illegible]

Instead, we use the *regula falsi* method

- credited to Babylonian Mathematics
- they had also invented sexagesimal (base 60) numeral system
- aka false position method
  1. Using 2 function values, find the abscissa crossing (as in the *secant* method)
  2. Adjust bounds to keep the root inside
- What we do: after first iteration $t_2 = t_1 + \Delta t(t_1)$
  1. use *regula falsi* if $\Delta t(t_1) * \Delta t(t_2) < 0$
  2. otherwise use $t_2 + \Delta t(t_2)$
  3. clamp Δt values to [-0.5, 0.5] interval (it helps in ray || cone situations)
  4. switch to $(t_i + t_{i-1})/2$ every 4th iteration

OptiX triX

From: Detlef Roettger
Sent: Friday, October 27, 2017 2:12 AM
To: Alexander Reshetov <areshetov@nvidia.com>
Subject: RE: NVIDIA-OptiX-SDK-5.0.0-DEV-win64.exe

Your RSVI curve primitives look better than anything I've seen in papers so far. Very nice!
Are the different colors indicating separate AABBs or just visualizing the proper t interpolant along the curve?

For additional performance analysis with OptiX you can make use of an OptiX developer build and look
at which device programs used how many clocks.
Normally intersection programs are inlined and the clocks of that appear inside the traversal as well,
but you can switch off inlining for that with another OptiX knob.

The procedure is simple:
- Create an empty file named optix.props next to your OptiX DLL.
- Run your application, end it.
- Now the OptiX developer build should have written all available knobs into that optix.props file as comments (~43 kB)
- Open it in an editor and search for the knob stats.timeVpcs uncomment and set it to 1.
- If you start your OptiX program now from a command line, OptiX will print out a table per launch (to stderr by default) with information about how many rays you shot, how many clocks were used and which program spent how many of them in absolute values and percentages.
The last column in that table shows the occupancy where 32 is perfect.

My non-default **optix.props** settings

`acceleration.bvh.traversal_cost`	`0.4`
`deviceManager.forceSmVersion`	`50`
`log.colored`	`0`
`megakernel.loadBalancer.memoryWeight`	`0.8`
`megakernel.loadBalancer.smWeight`	`0.2`
`megakernel.register.attributeSwitch`	`1`
`megakernel.register.currentTmax`	`1`

also using `context->createAcceleration("Bvh8")`

Performance

400K curves, 145M rays/s @ Titan Xp

1K snakes, 250M rays/s

a. 58537 curves 412M rays/s

b. 139506 curves 400M rays/s

c. 729628 curves 322M rays/s

d. 271902 curves 515M rays/s

e. 756453 curves 531M rays/s

A Titan Xp performance for the different yarn models [Wu and Yuksel 2017; Yuksel et al. 2012] for one primary and four ambient occlusion rays at 1000×1000 screen resolution

a. 58537 curves

412M rays/s

urves

s/s

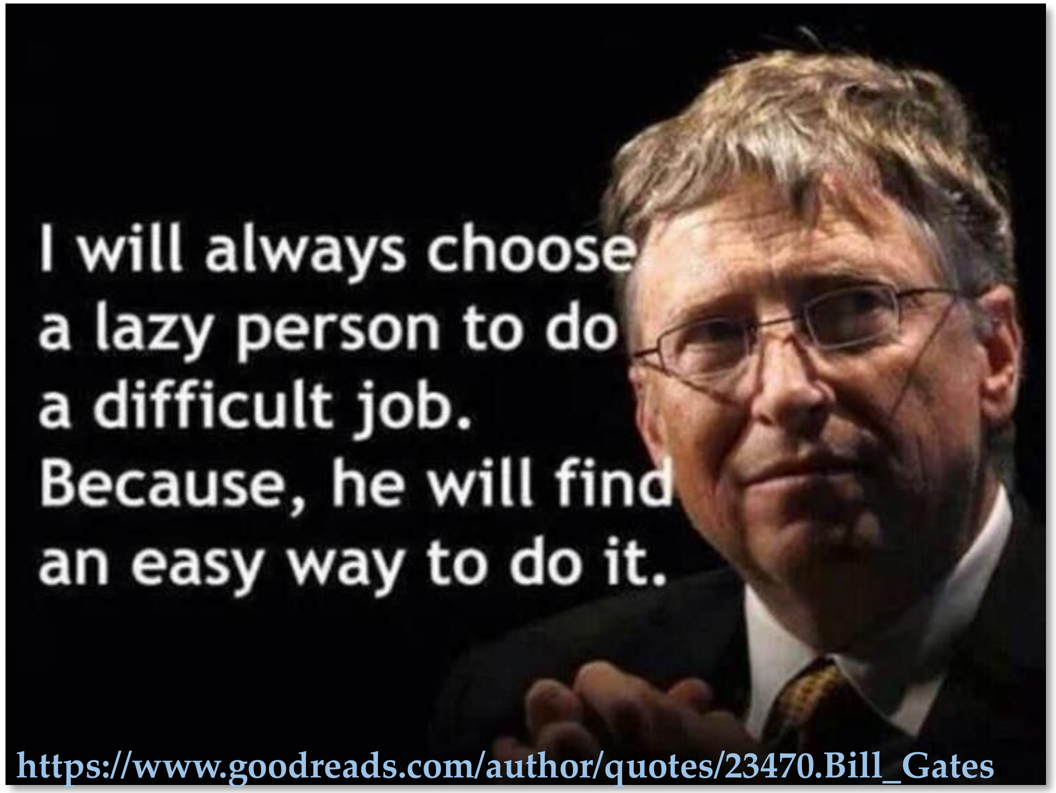
e. 756453 curves

531M rays/s

A Titan Xp perform *Yuksel et al. 2012]*
for one primary and *ution*

Limitations

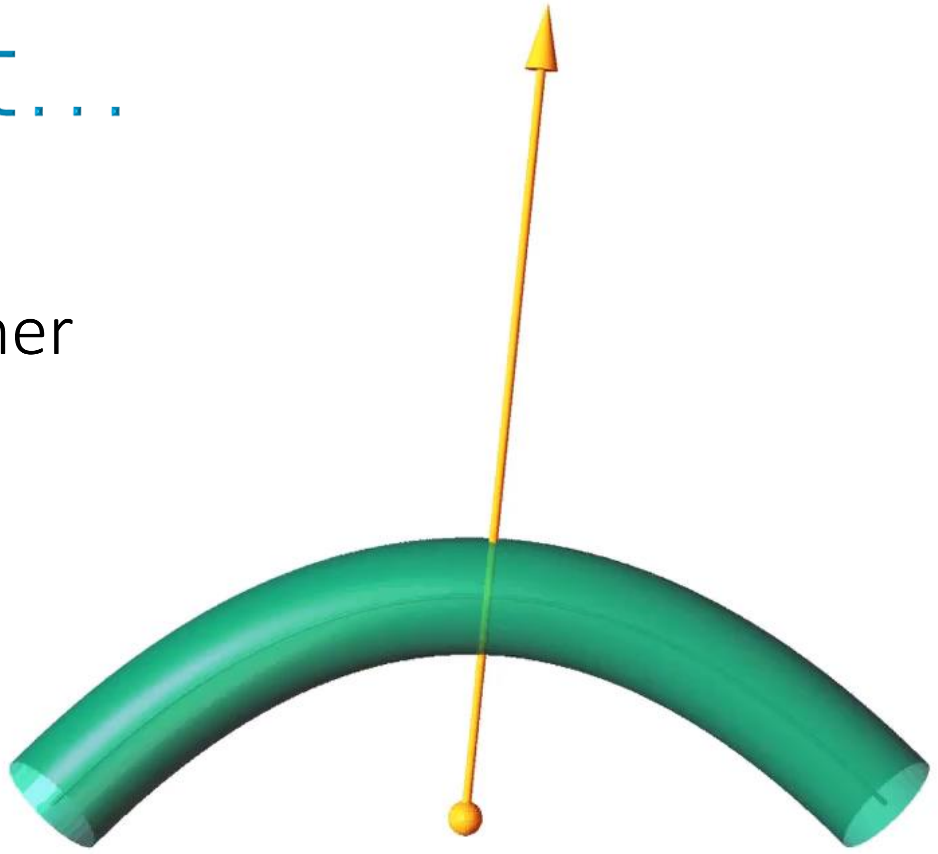
- It is all applicable only for ray-tracing problems (“reverse rasterization”)
- By itself, “finding roots of Δt(t)” is just an algebra-speak for “finding ray – surface intersections”
- But we want to do it lazily…
- … and it might cause problems

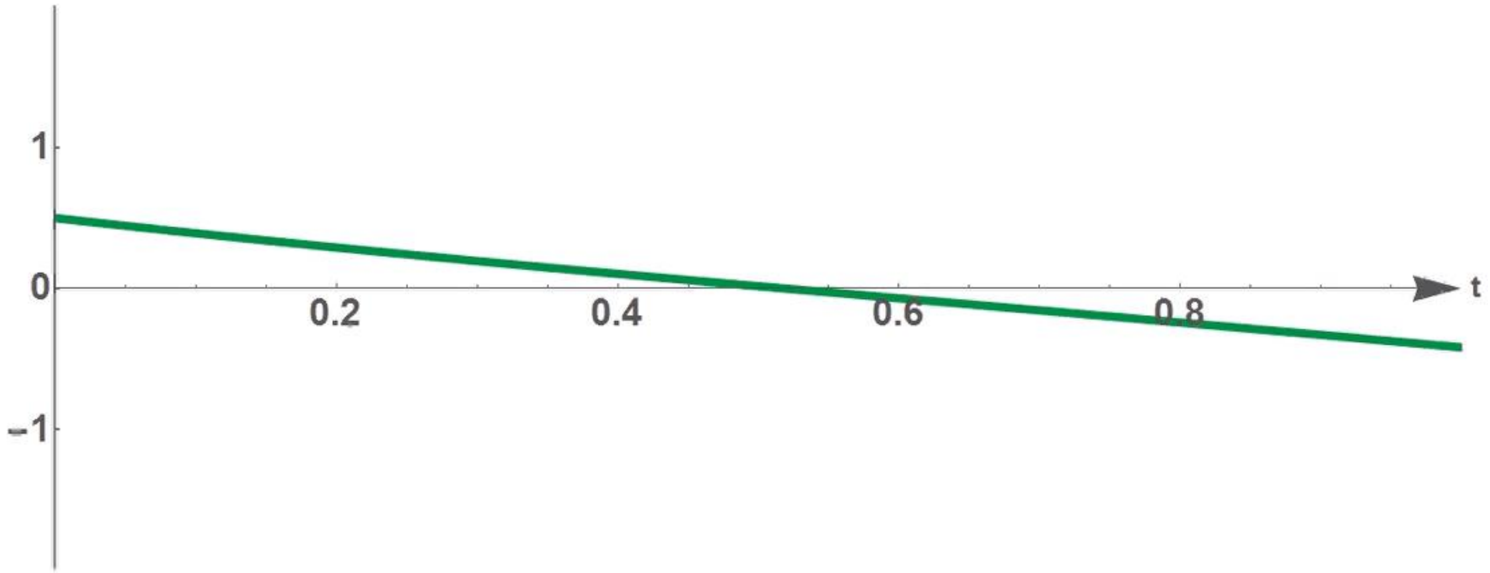

I will always choose a lazy person to do a difficult job. Because, he will find an easy way to do it.

https://www.goodreads.com/author/quotes/23470.Bill_Gates

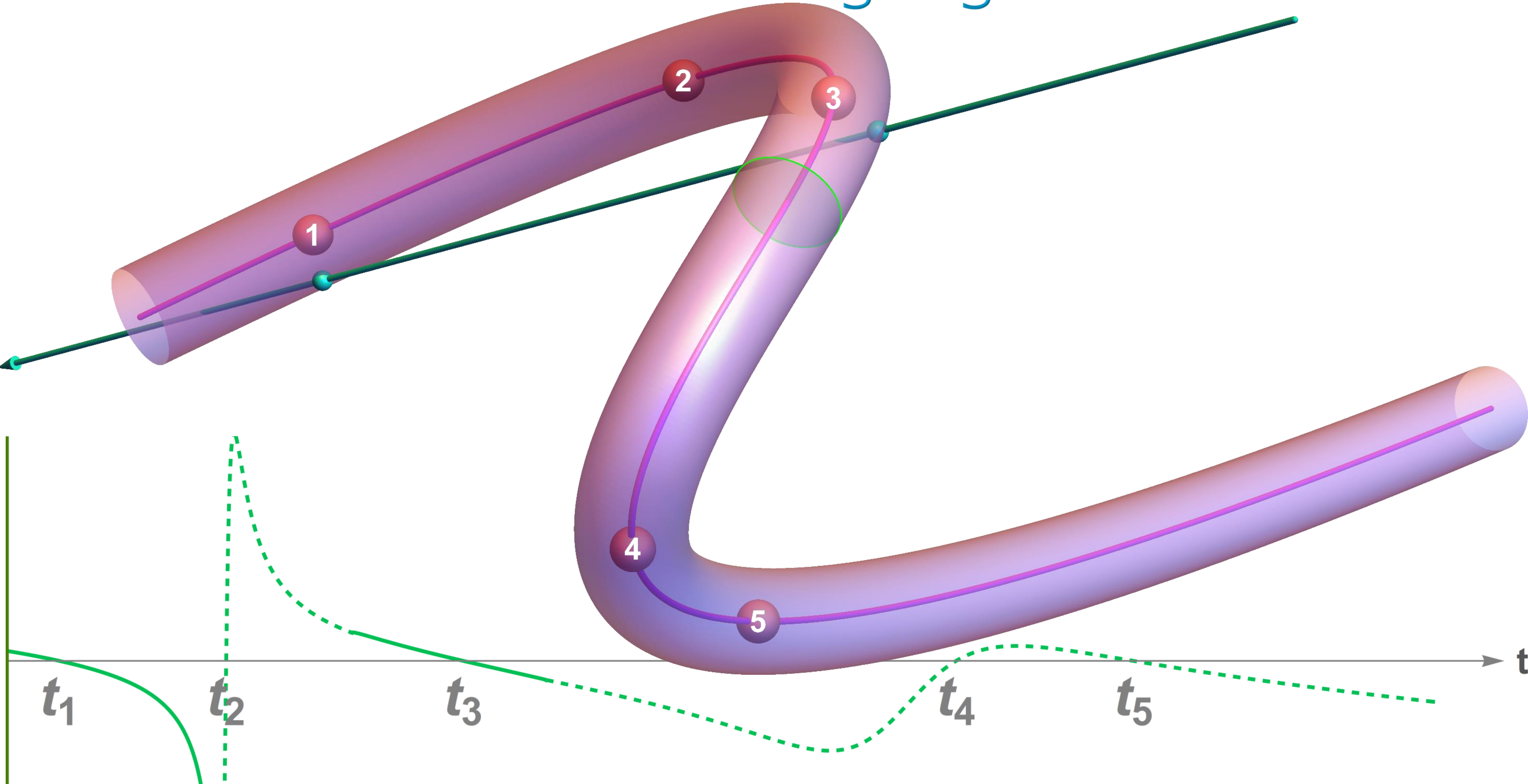
Another experiment…

- $\Delta t(t)$ is almost linear;
  root $t = 0.5$ can be found starting at either
  $t = 0$ or $t = 1$
- Phantom values near
  $t = 0$ or $t = 1$; $\Delta t(t)$ is still $\approx$ linear
- root $t = 0.5$ got hidden by the phantom
  values near $t = 0$ or $t = 1$

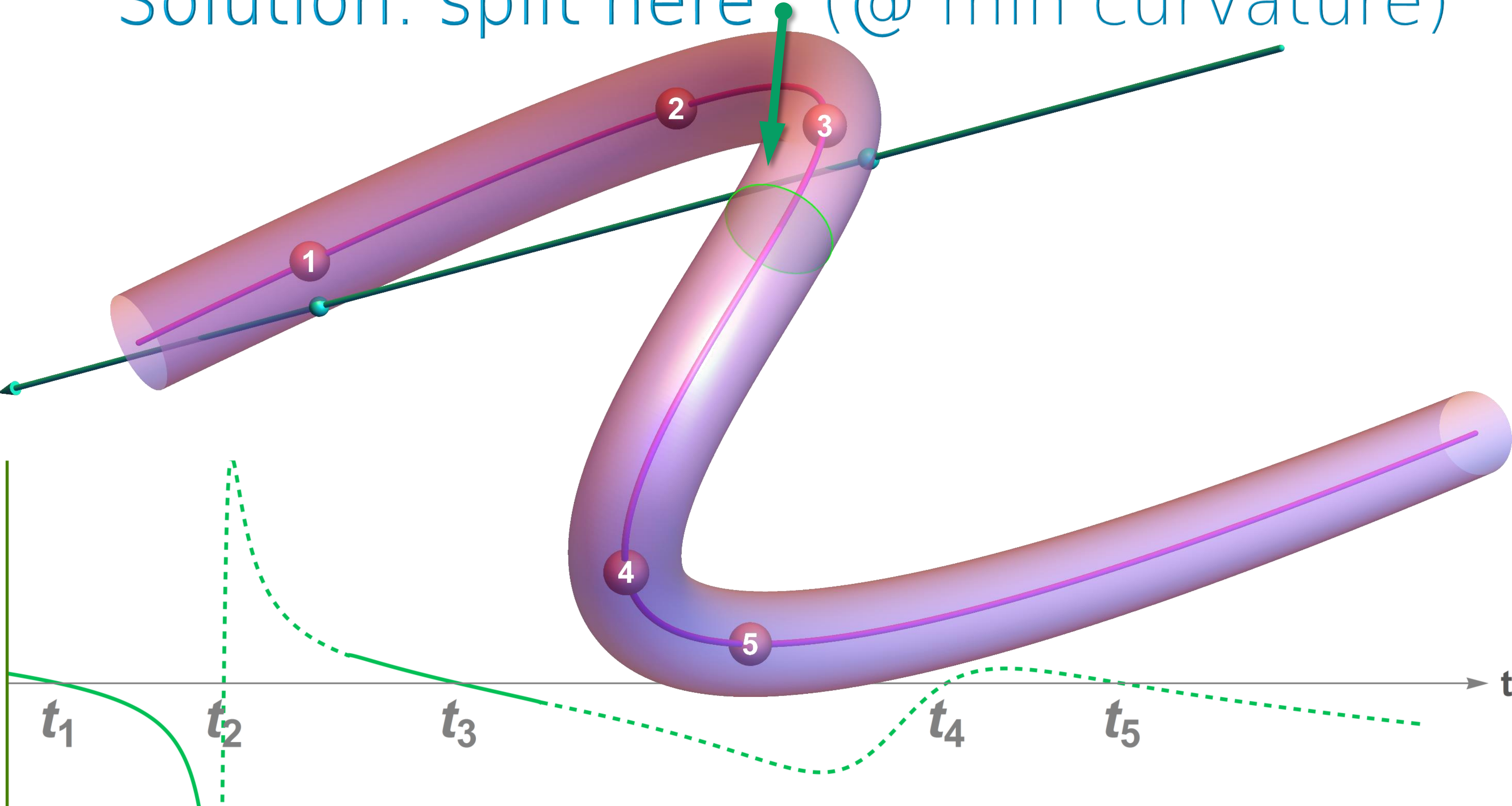


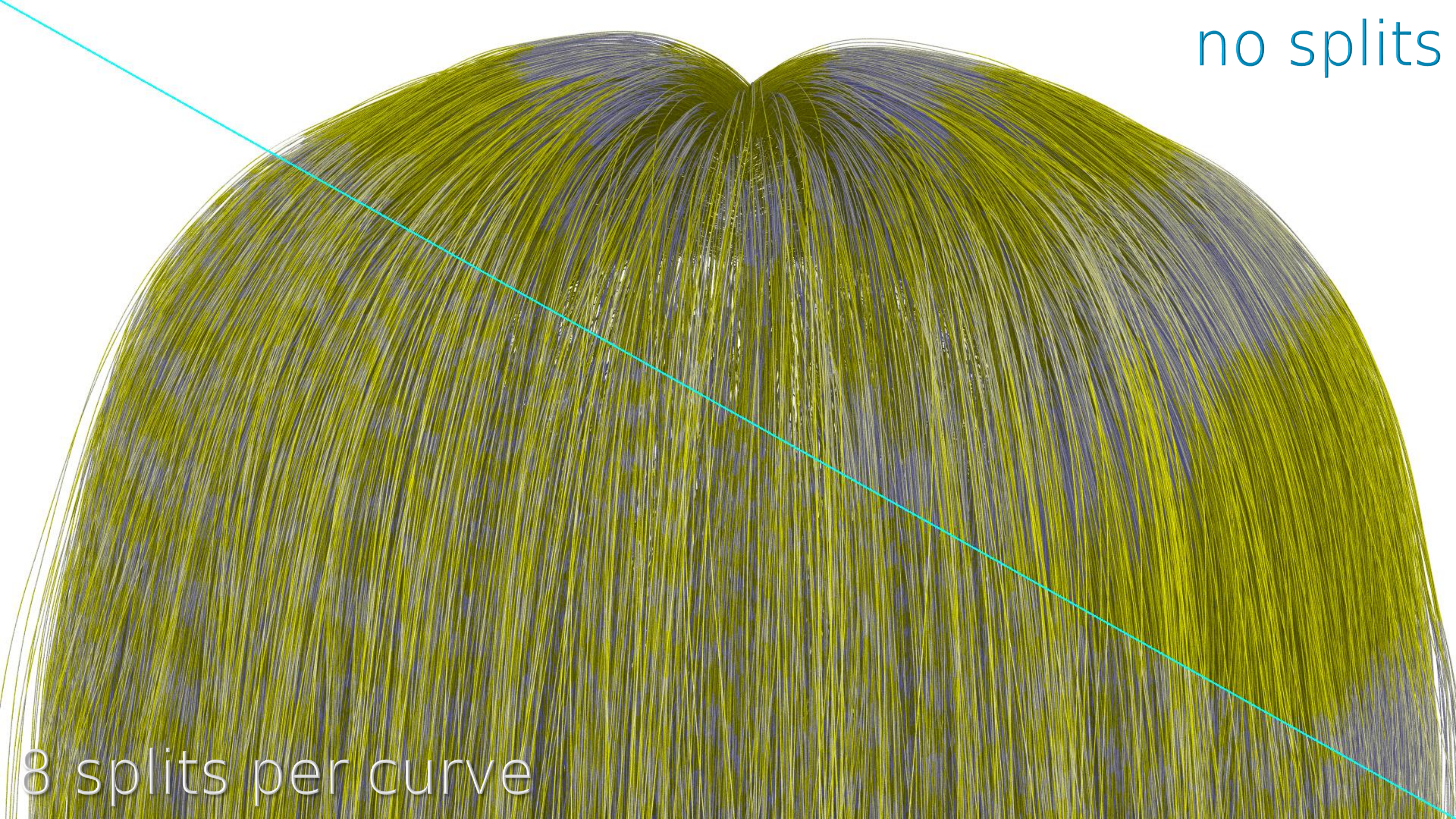


Problems with zigzag curves



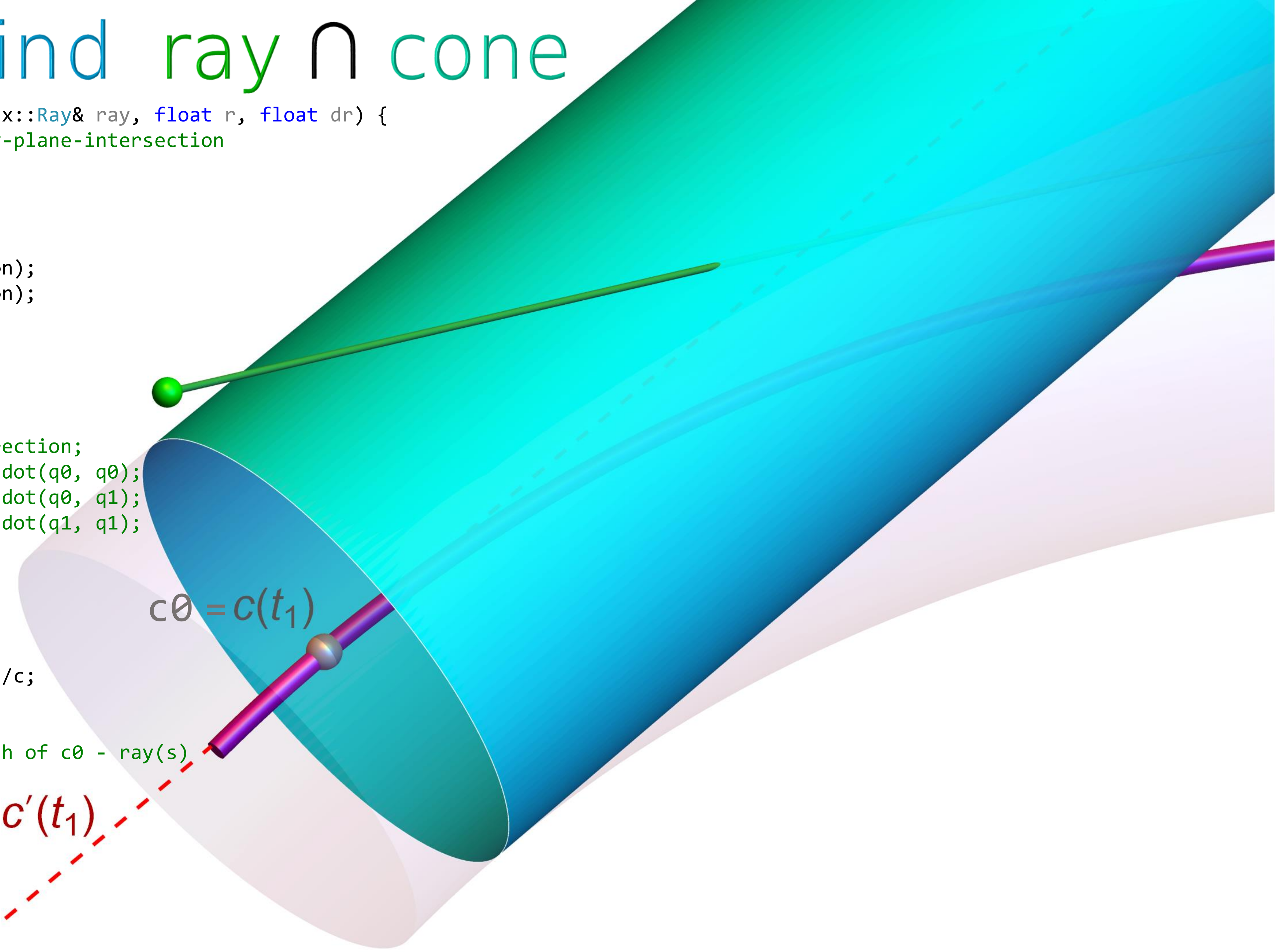
Solution: split here (@ min curvature)




no splits
8 splits per curve

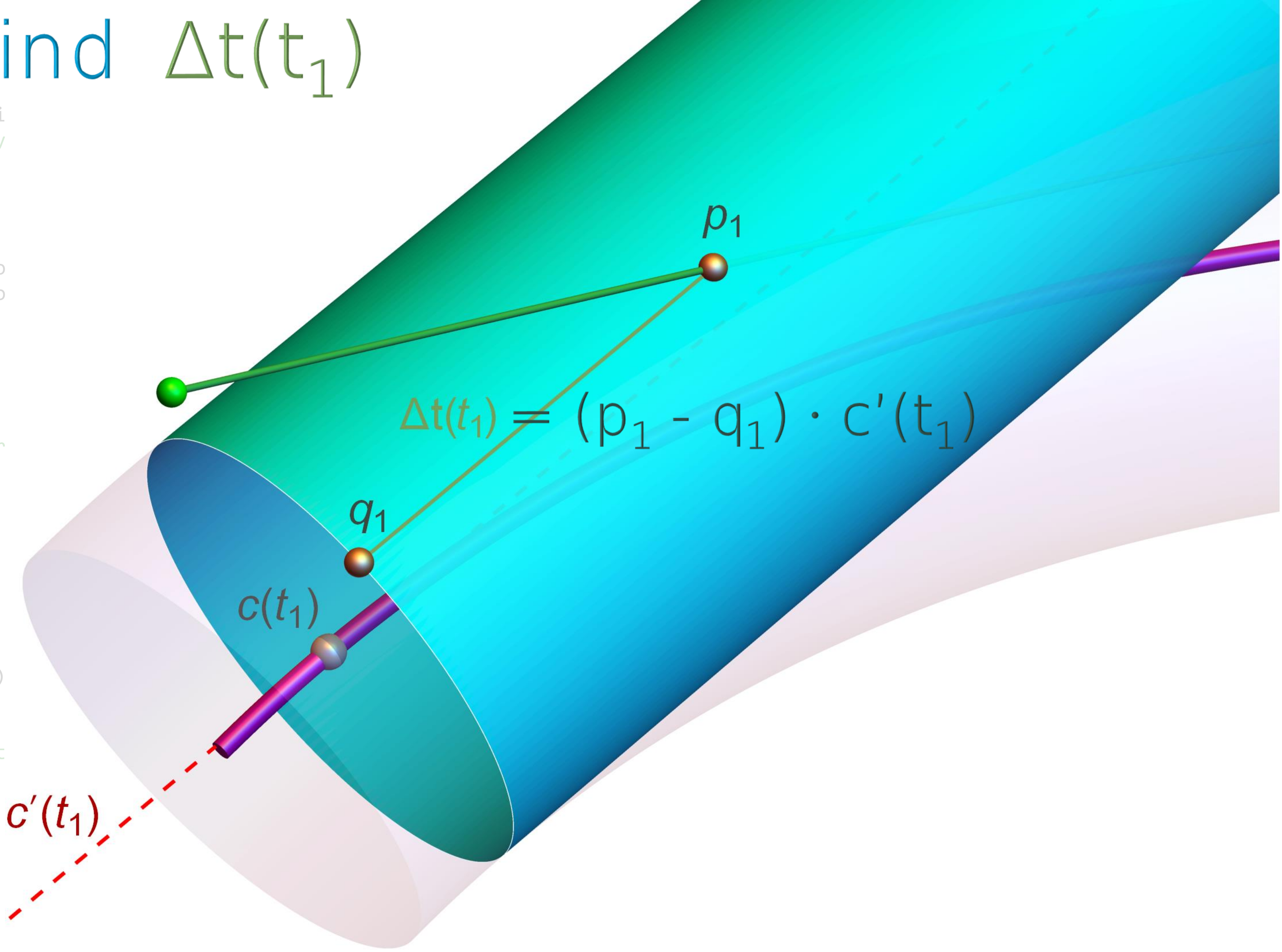
Step 1/3: find ray ∩ cone

```
inline bool intersectCone(const optix::Ray& ray, float r, float dr) {
  // d = q0 + q1*s = (c0+cd*t) - ray-plane-intersection
  float3   cr = c0 - ray.origin;
  float  cdcd = dot(cd, cd);
  float  crcd = dot(cr, cd);
  float  crcr = dot(cr, cr);
  float  crrd = dot(cr, ray.direction);
  /* */  cdrd = dot(cd, ray.direction);
  float cdrdn = cdrd/cdcd;
  float crcdn = crcd/cdcd;
  r  = r - dr * crcdn;
  dr = dr * cdrdn;
  //float3 q0 = cr - cd * crcdn;
  //float3 q1 = cd * cdrdn - ray.direction;
  float q00 = crcr - crcd*crcdn; // dot(q0, q0);
  float q01 = cdrd*crcdn - crrd; // dot(q0, q1);
  float q11 = 1 - cdrd*cdrdn;    // dot(q1, q1);

  float a = q00 - r*r;
  float b = q01 - r*dr;
  float c = dr*dr - q11;
  float det = b*b + a*c;
  s = (b + (det < 0? 0 : sqrt(det)))/c;
  dt = cdrdn * s - crcdn;

  // Compute |cr - s rd|, i.e. length of c0 - ray(s)
  dc = crcr - 2*crrd*s + s*s;

  sp = crcd/cdrd;
  dp = crcr - 2*crrd*sp + sp*sp;

  return det > 0;
}
```

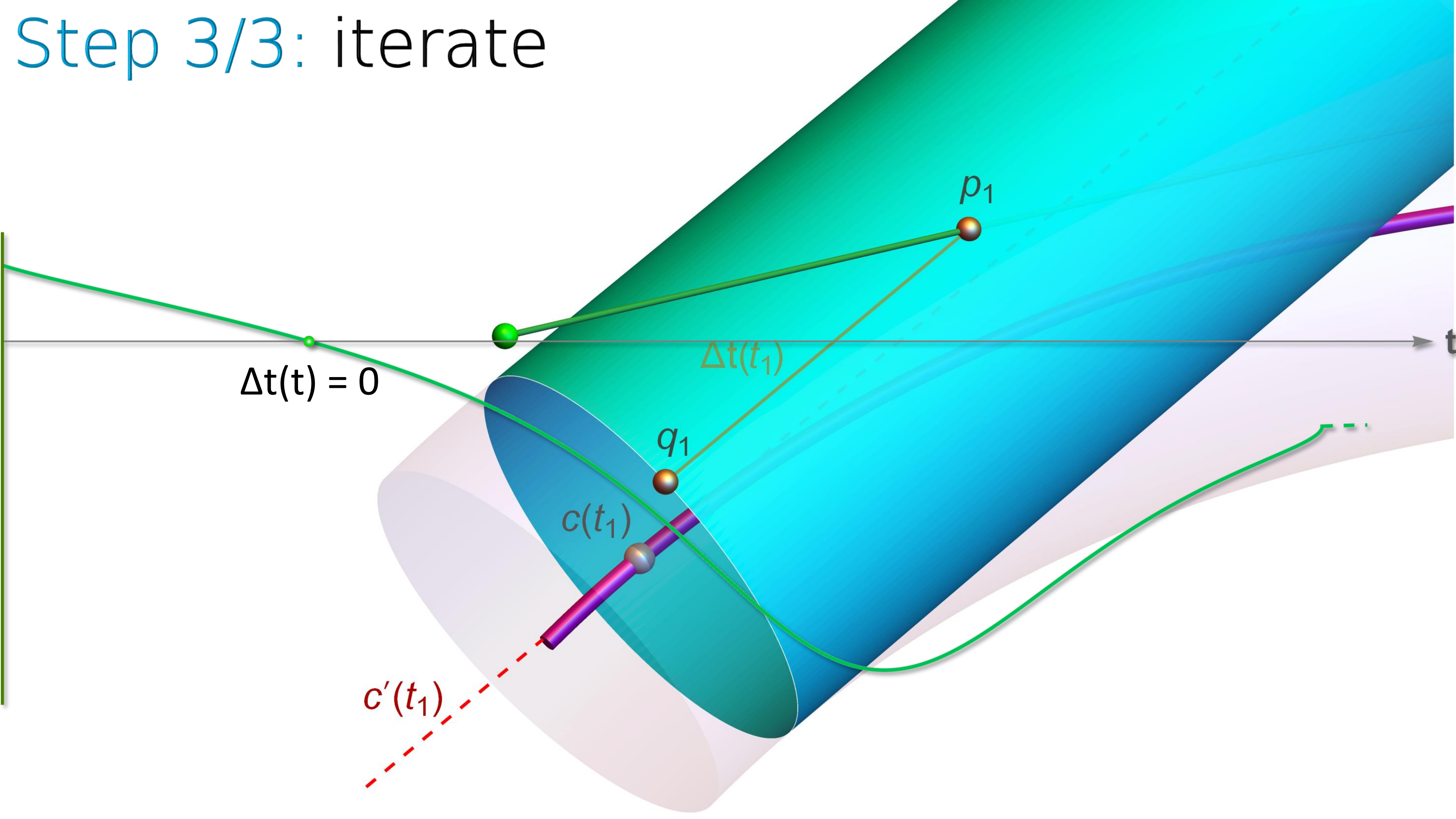


Step 2/3: find $\Delta t(t_1)$

```
inline bool intersectCone(const opti
  // d = q0 + q1*s = (c0+cd*t) - ray
  float3   cr = c0 - ray.origin;
  float  cdcd = dot(cd, cd);
  float  crcd = dot(cr, cd);
  float  crcr = dot(cr, cr);
  float  crrd = dot(cr, ray.directio
  /* */  cdrd = dot(cd, ray.directio
  float cdrdn = cdrd/cdcd;
  float crcdn = crcd/cdcd;
  r  = r - dr * crcdn;
  dr = dr * cdrdn;
  //float3 q0 = cr - cd * crcdn;
  //float3 q1 = cd * cdrdn - ray.dir
  float q00 = crcr - crcd*crcdn; //
  float q01 = cdrd*crcdn - crrd; //
  float q11 = 1 - cdrd*cdrdn;    //

  float a = q00 - r*r;
  float b = q01 - r*dr;
  float c = dr*dr - q11;
  float det = b*b + a*c;
  s = (b + (det < 0? 0 : sqrt(det)))
  dt = cdrdn * s - crcdn;

  // Compute |cr - s rd|, i.e. lengt
  dc = crcr - 2*crrd*s + s*s;

  sp = crcd/cdrd;
  dp = crcr - 2*crrd*sp + sp*sp;

  return det > 0;
}
```



Step 3/3: iterate



The topics covered in this presentation

- Killer drones
- Newton’s pets
- Bill Gate’s pet peeves
- Daniel Day-Lewis’ last movie
- Bender Bending Rodríguez’ vocation
- Control systems for particle accelerators
- Star constellations
- Maritime science
- Rendering of flying saucers; sausage; snakes; hair/fur; yarn
- World Cup 2018
- Germany’s population growth
- Hardline rock-n-roll
- Zombies invasion

pensive things

We saw this whole zombie thing coming from a mile away…
we already have safe houses and are stocked up…

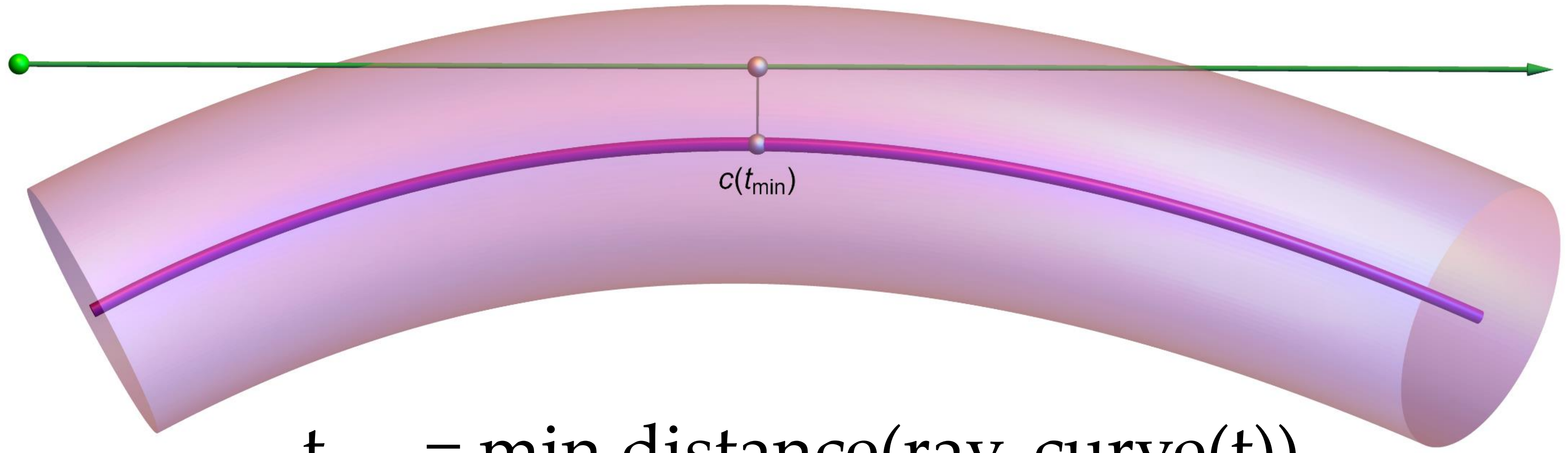
#zombielove

http://www.purevolume.com/new/phathom

Sagitta!

```
inline bool intersectCylindricalEnclosure(const optix::Ray& ray, float2& limits) const {
   // cylinder is defined by axis acyl, radius^2 = rcyl and point on the axis pcyl
   float3 nn = cross(ray.direction, acyl); // orthogonal to both
   float3 r2c = ray.origin - pcyl;
   float distance_from_ray_to_cylinder_axis = dot(r2c, nn);
   float innlen2 = 1.0f/dot(nn, nn);

   float d2 = distance_from_ray_to_cylinder_axis * distance_from_ray_to_cylinder_axis;
   float sagitta = rcyl - d2 * innlen2;

   // compute distance from ray.origin to the closest point between the ray and acyl
   float3 o2c = cross(r2c, acyl);
   float po2c = dot(o2c, nn);
   limits.x = limits.y = -po2c * innlen2;
   // now reproject 2D sagitta (in the plane orthogonal to the tangent) back into 3D
   sagitta  = sagitta > 0? sqrt(sagitta * innlen2) : 0;
   limits.x = optix::fmaxf(limits.x - sagitta, ray.tmin);
   limits.y = optix::fminf(limits.y + sagitta, ray.tmax);

   return limits.y > 0 && limits.x < limits.y;
}
```

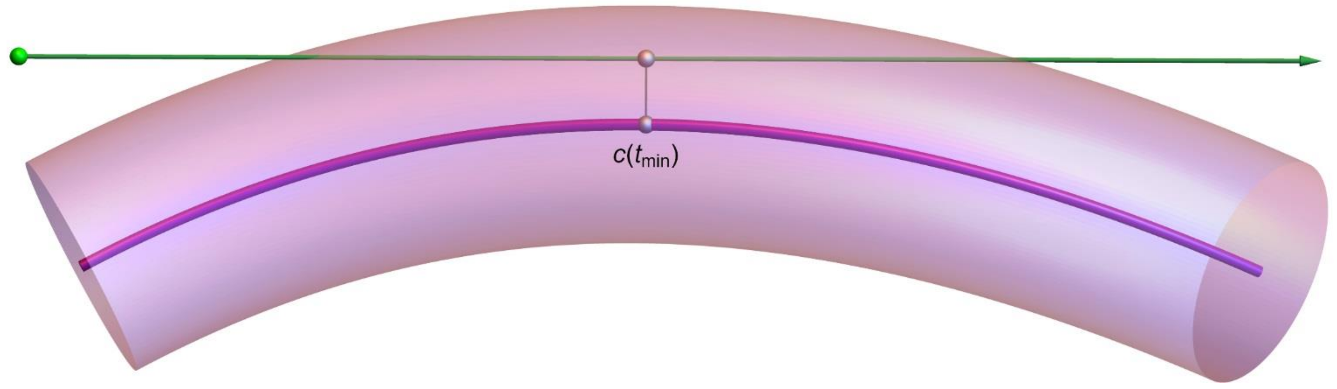
state-of-the-art = closest-point-of-approach



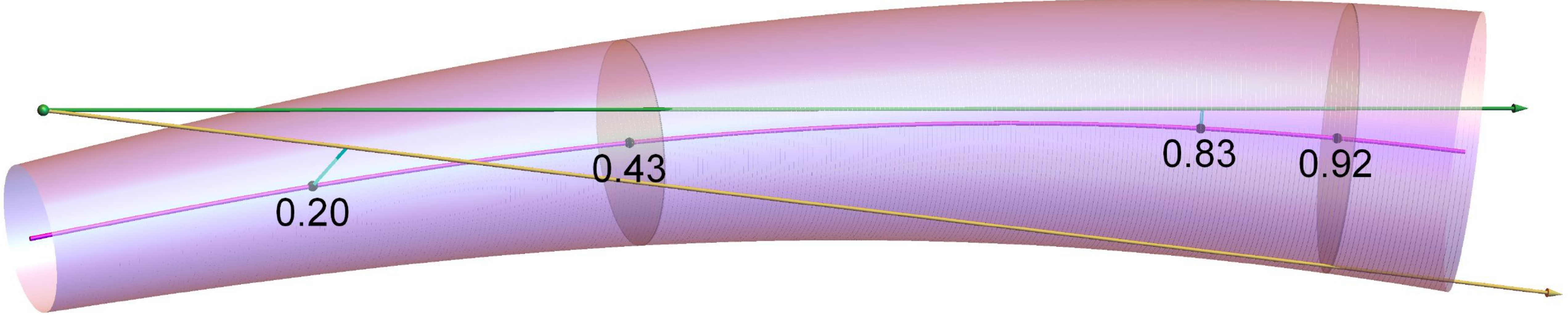
$$t_{\min} = \min_{t \in [0,1]} \text{distance}(\text{ray}, \text{curve}(t))$$

Aye and nay for CPA

- Pros
  - Simple; in practice: adaptive linearization
  - Might be useful for the collision detection and repulsion
- Cons
  - Non-physically based
  - What is the surface normal?
  - View-dependent results
  - Artificial high-order frequencies

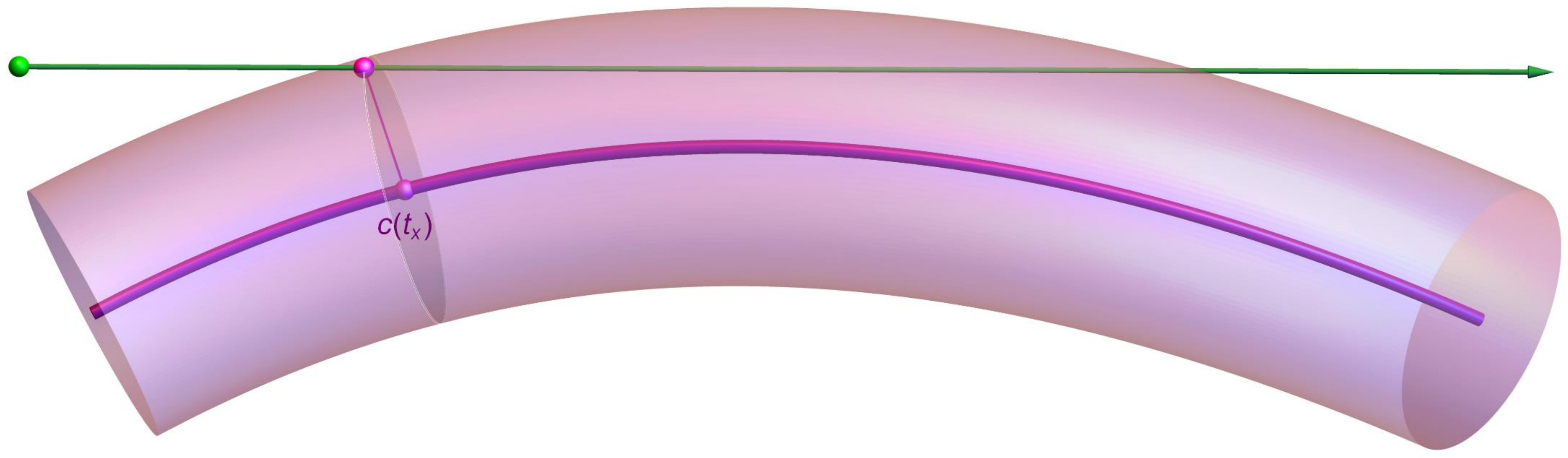


When CPA fails



- CPA root at t = 0.2
- Distance[curve[0.2], ray] / radius[t] = 1.4
- Ray-swept volume intersection at t = 0.92

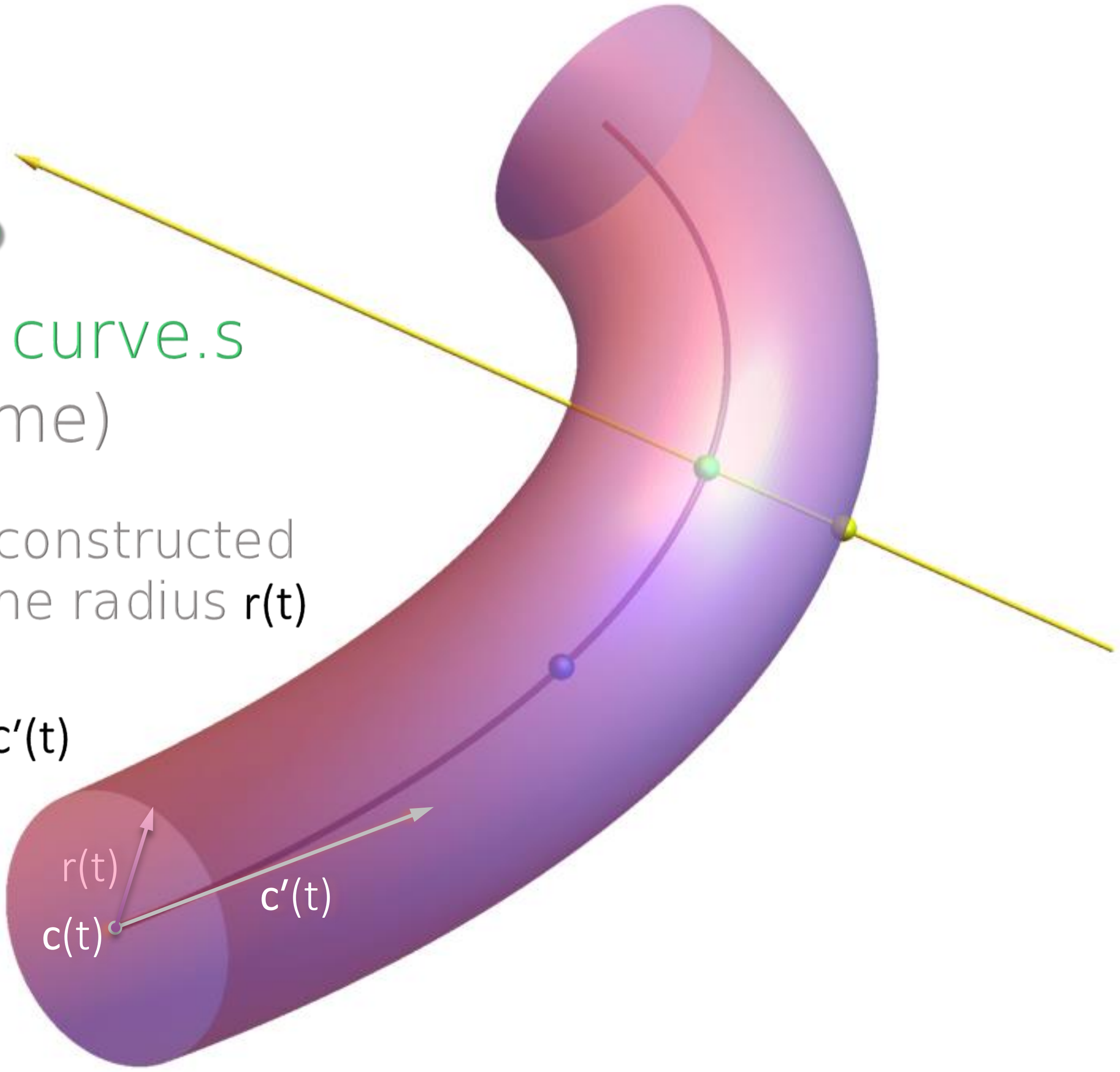
Instead, we search for ray ∩ swept volume



We want ● and ●

(defined by ray.t and curve.s
for ray ∩ swept volume)

Note: the swept volume is constructed by extruding a circle with the radius r(t) centered at c(t) in the plane orthogonal to c′(t) along the curve



🟡 and 🟢 properties

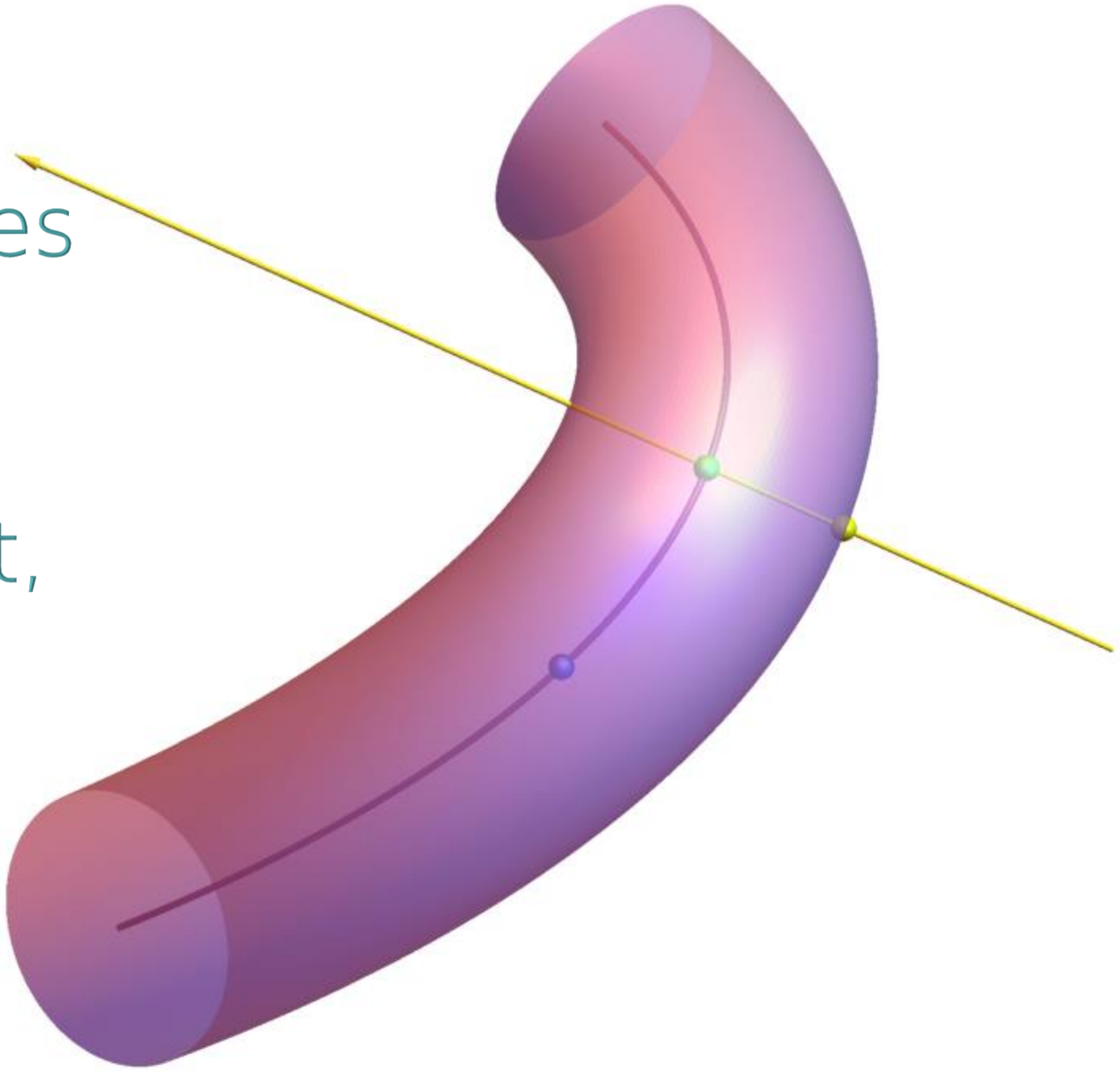
|🟡 – 🟢| == $r(t_x)$

(🟡 – 🟢) · $\mathbf{c}'(t_x)$ == 0

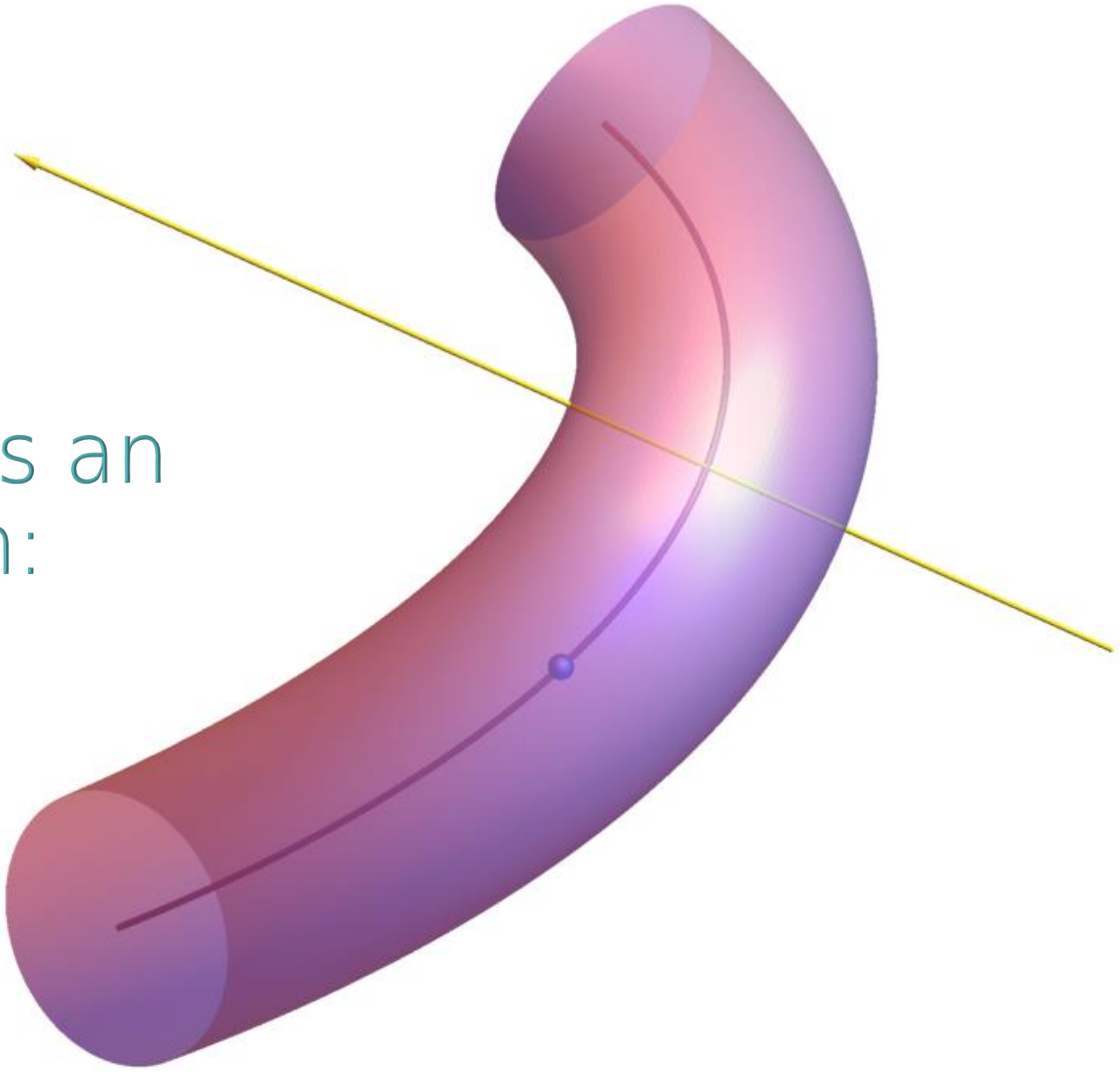
Once we found it,

🟡 is the hit point

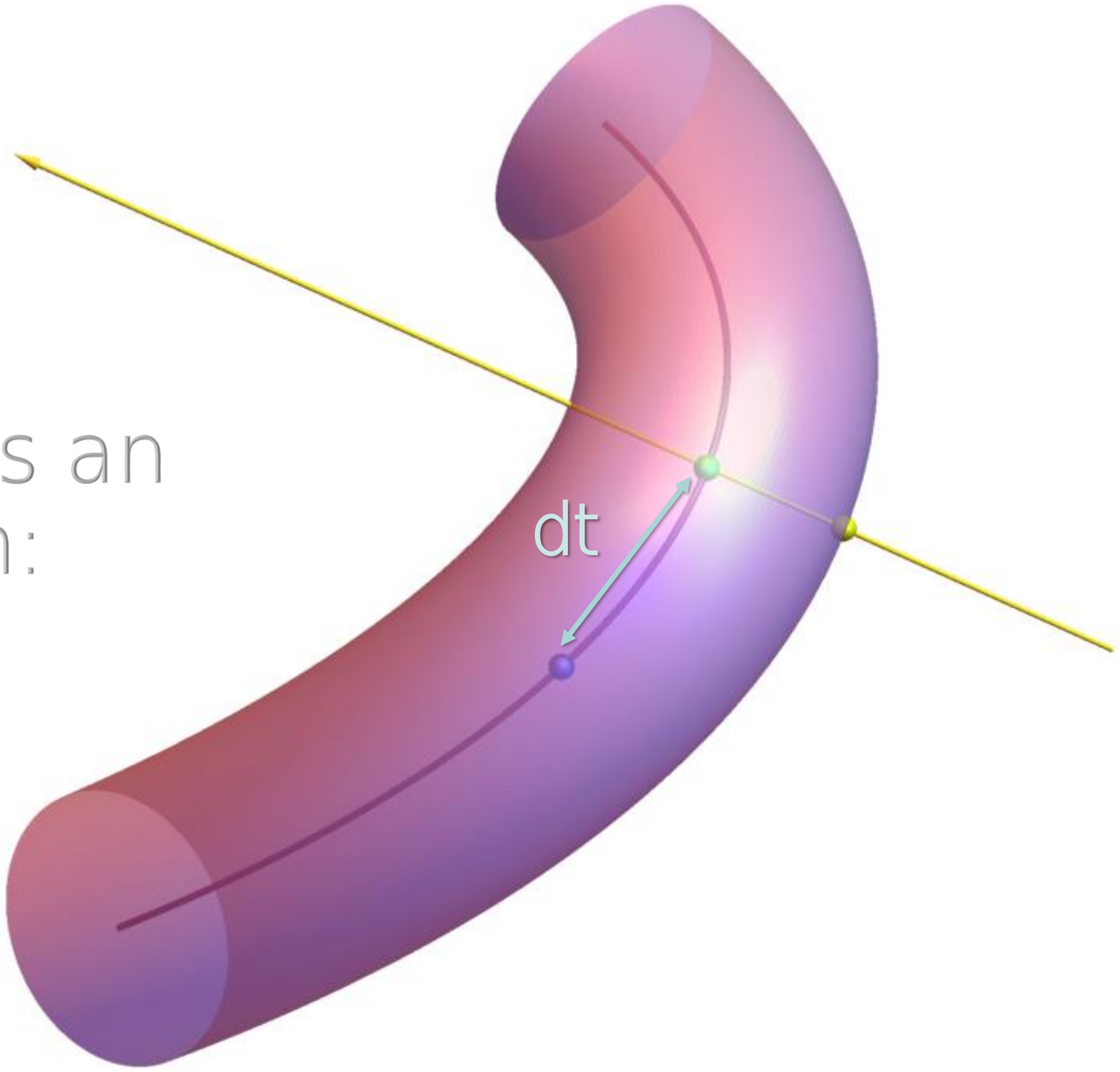
(🟡 – 🟢)/| 🟡 – 🟢 | gives the surface normal (for cylinder; more complex expression for other profiles)



Phantom xsector is an iterative algorithm: for any initial ●

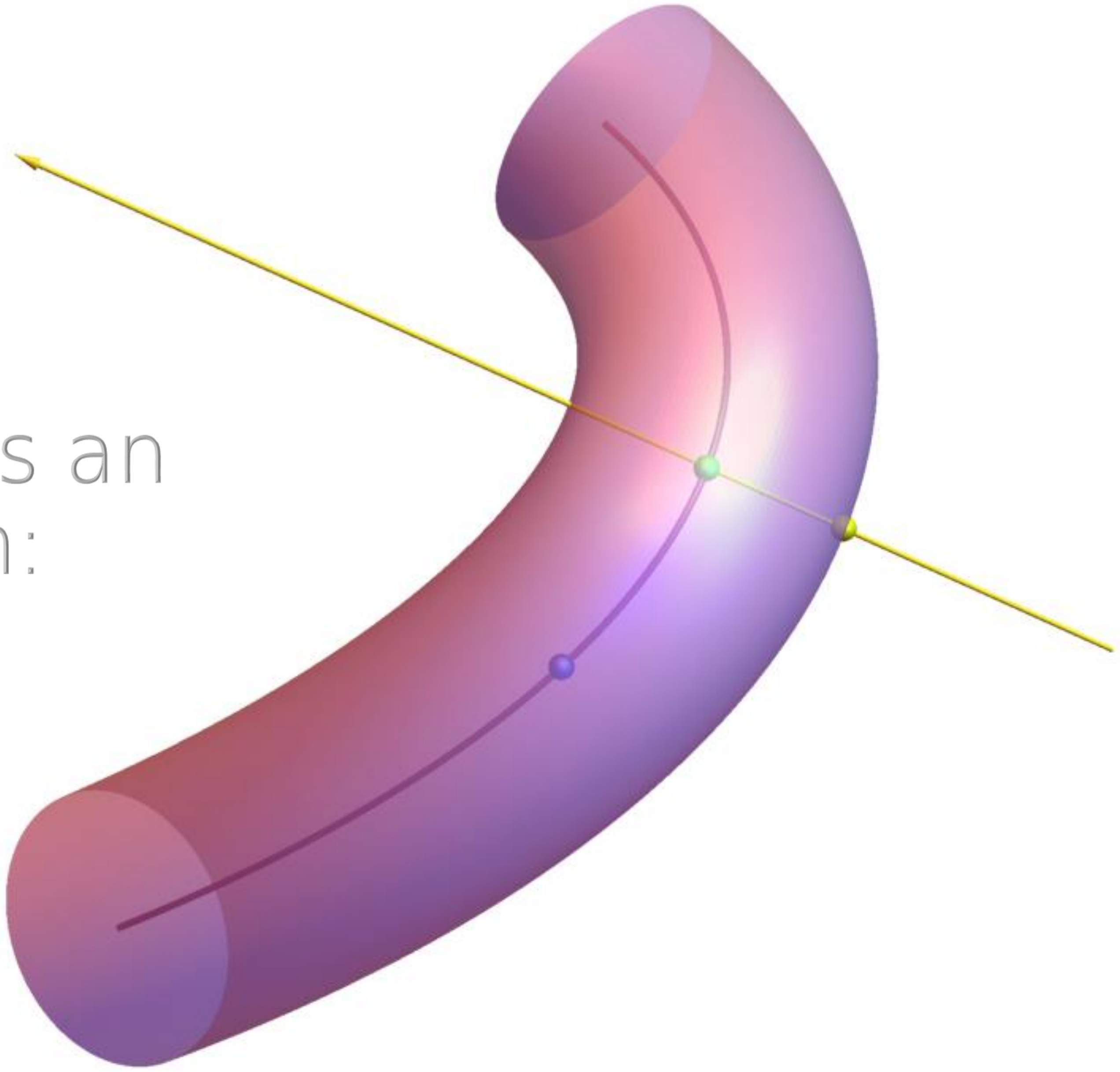


Phantom xsector is an iterative algorithm:
for any initial ●
we will find
(the approximate)
dt = ● - ●



Phantom xsector is an iterative algorithm: for any initial ● we will find (the approximate) dt = ● - ●

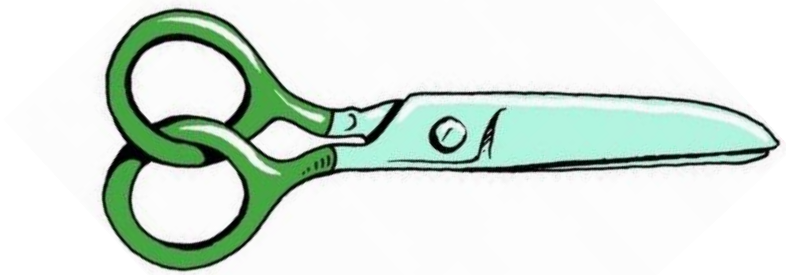
and also find ● as a side effect

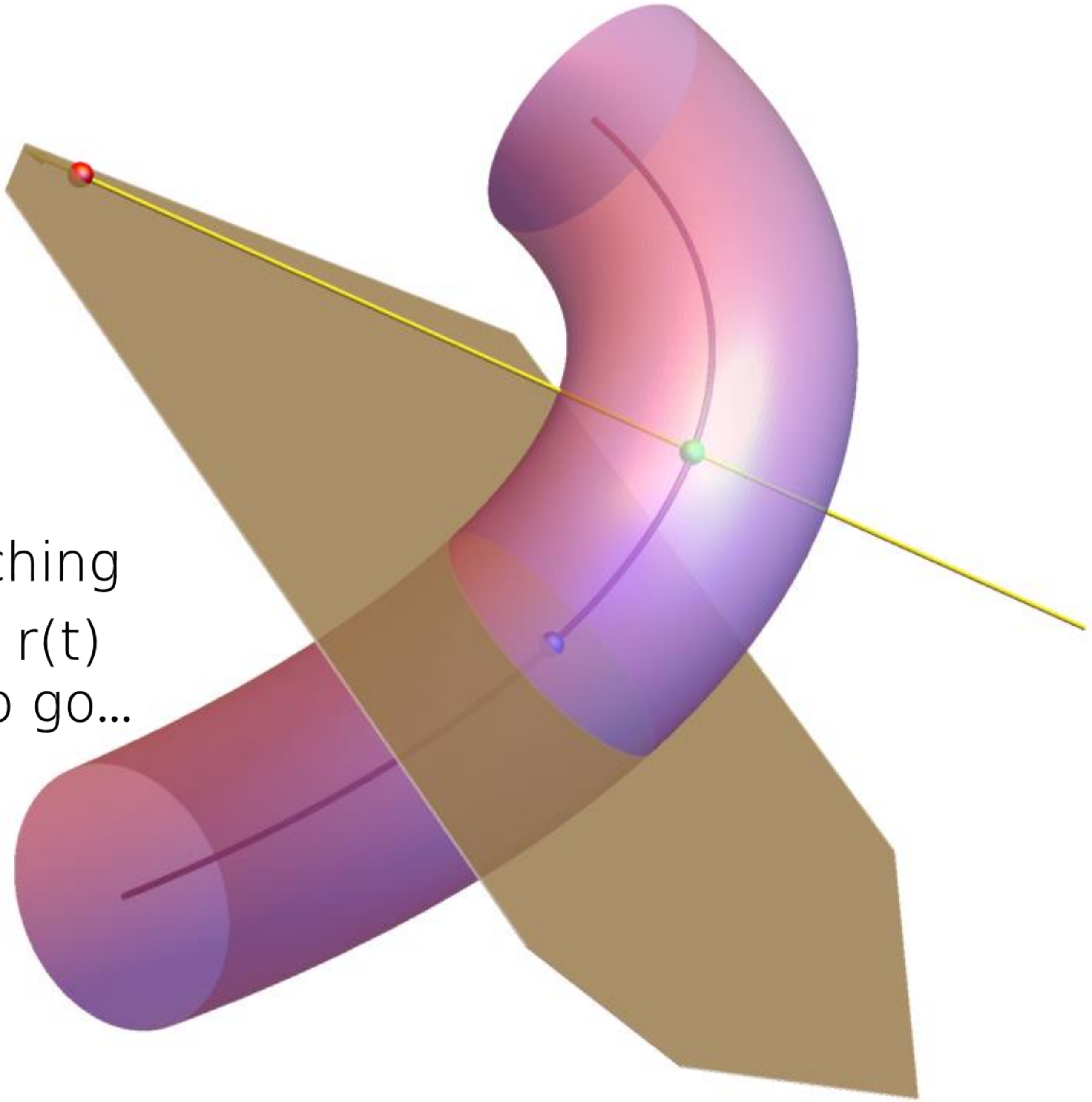


Because
(🟡 − 🟢) · **c′**(t_x) == 0 and
|🟡 − 🟢| == r(t_x)
@ ray ∩ volume,

finding 🔴 and then searching
for root(s) of |🔴 - 🟣| == r(t)
might seem like a way to go…

… yet it is a terrible idea



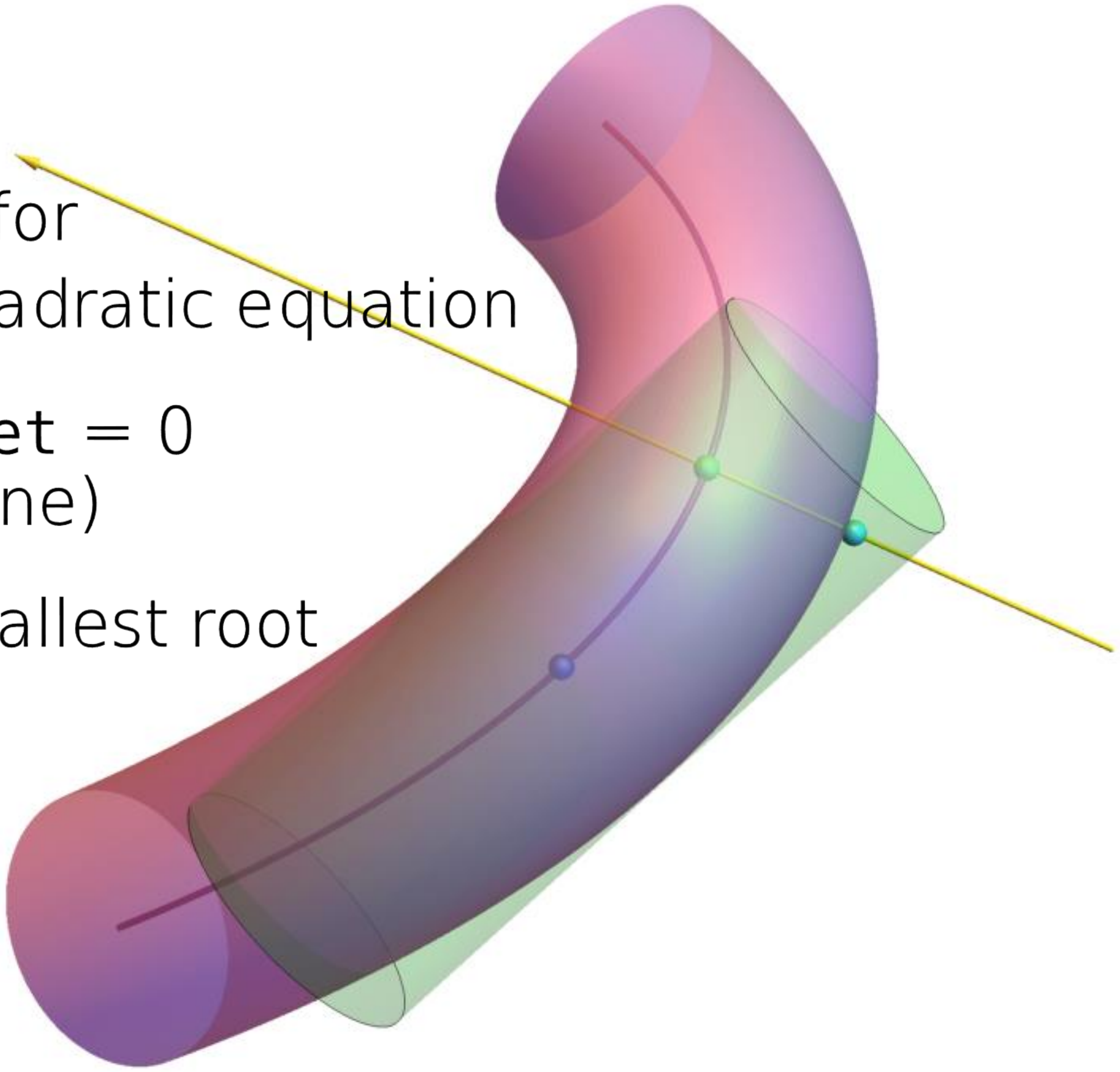


Instead, we will look for

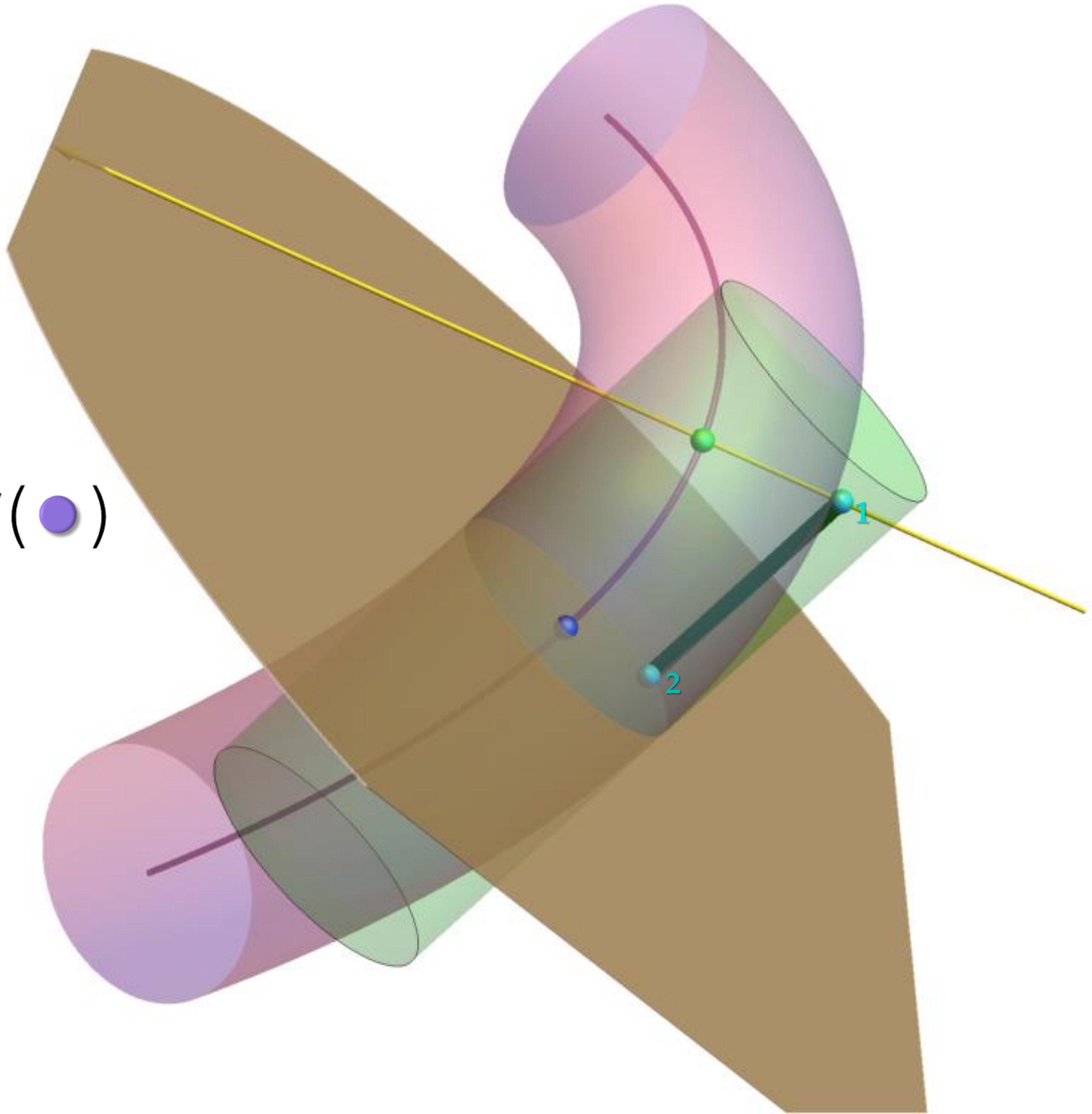
- ● (ray ∩ cone) ⇦ quadratic equation

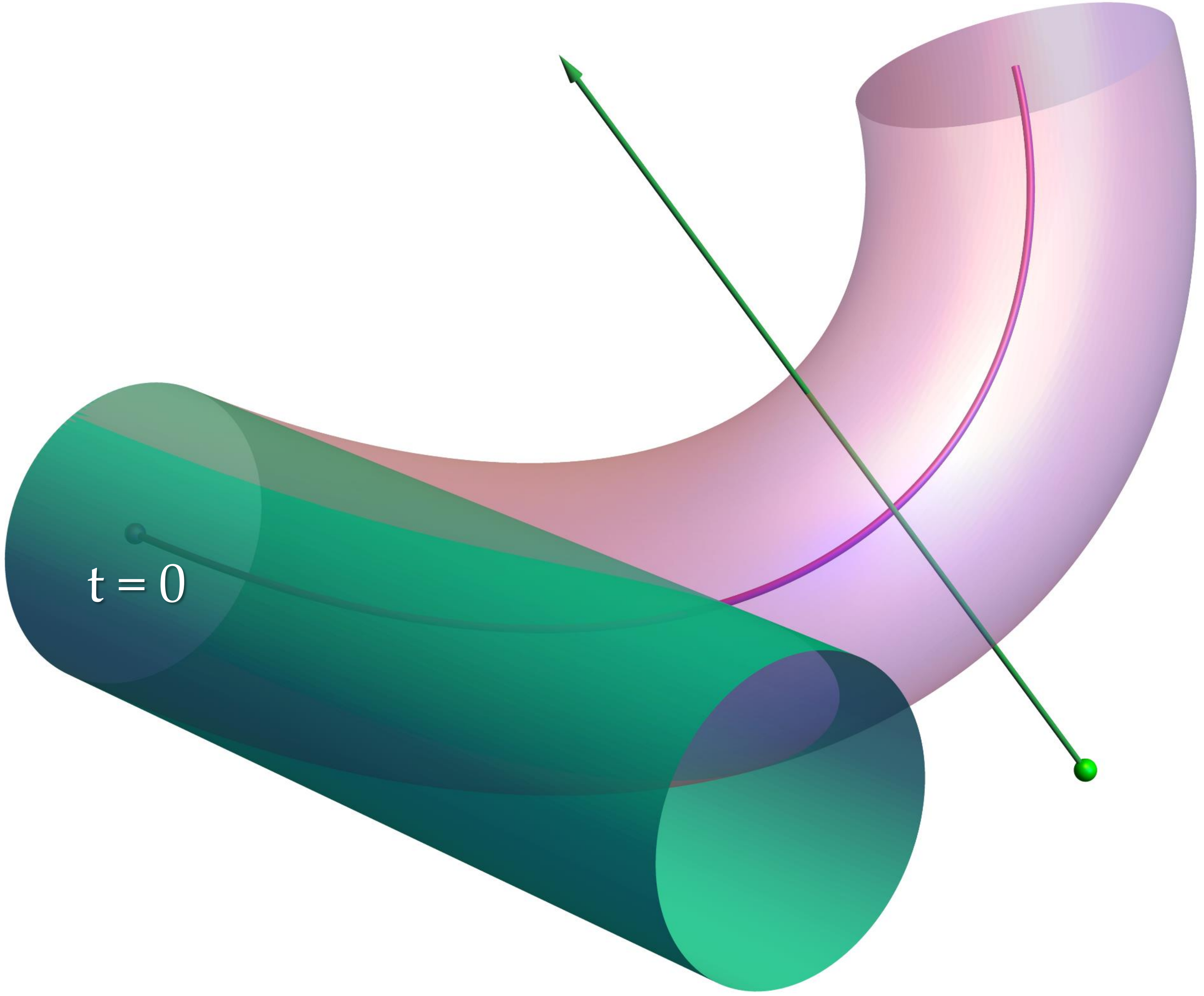
If **det** < 0, we'll set **det** = 0
(ray ∩ the padded cone)

We need only the smallest root



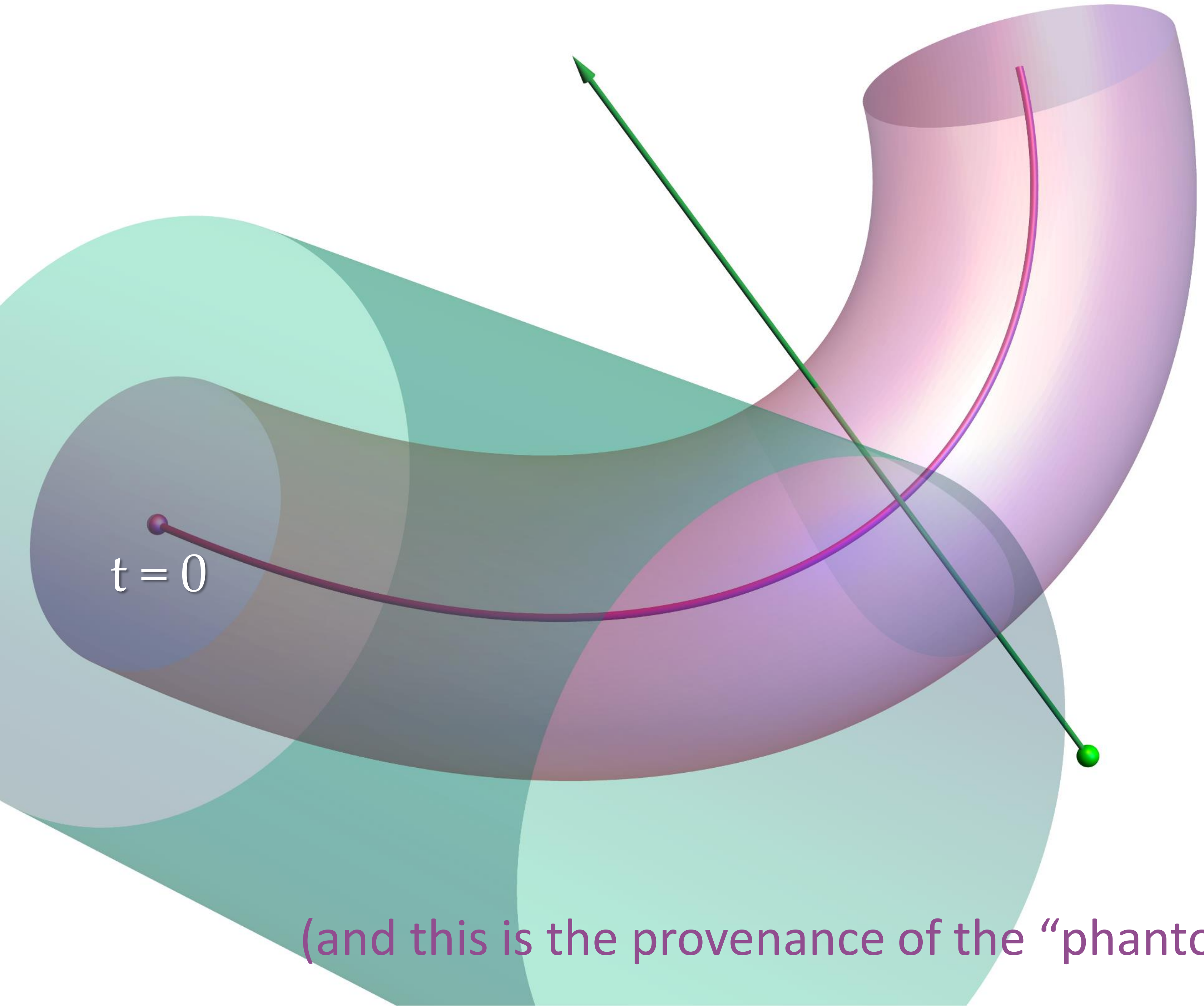
$$dt(\bullet) = (\bullet_1 - \bullet_2) \cdot c'(\bullet)$$





Example of dt function:

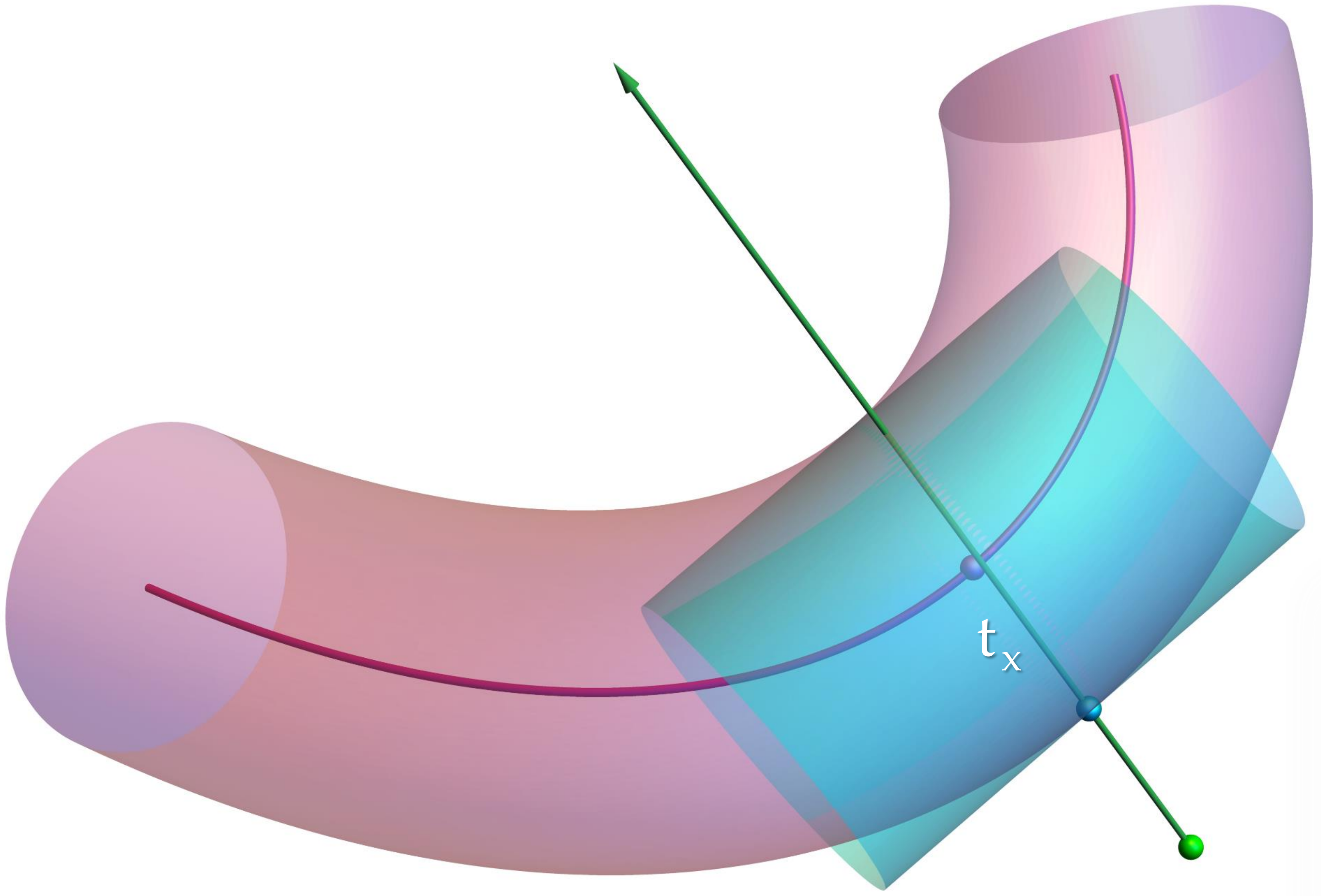
at t = 0, the ray does not intersect cone(t).



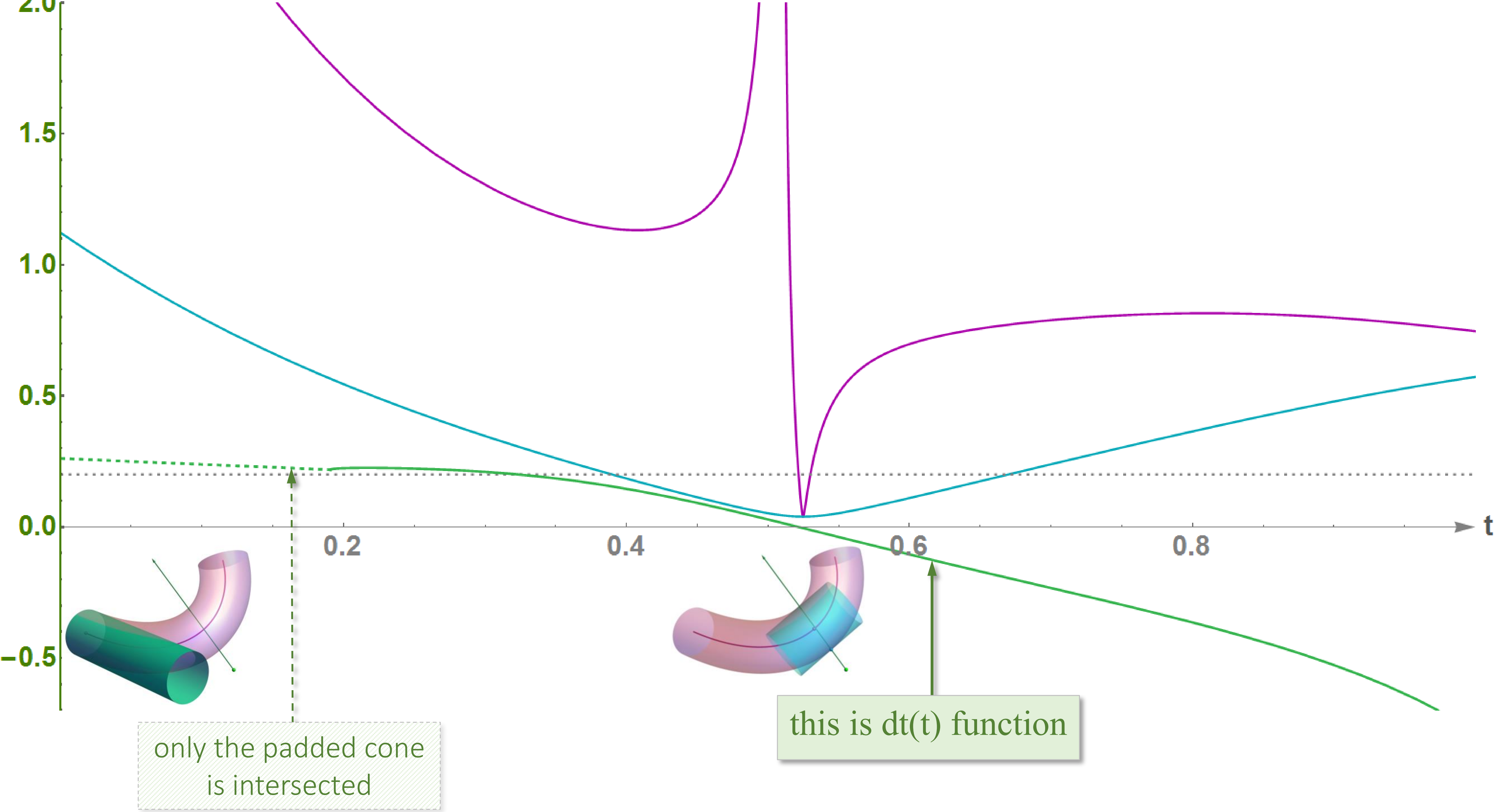
Example of dt function:

at t = 0, the ray does not intersect cone(t). Yet, it touches the padded cone, allowing to find dt(0)

(and this is the provenance of the “phantom” name)

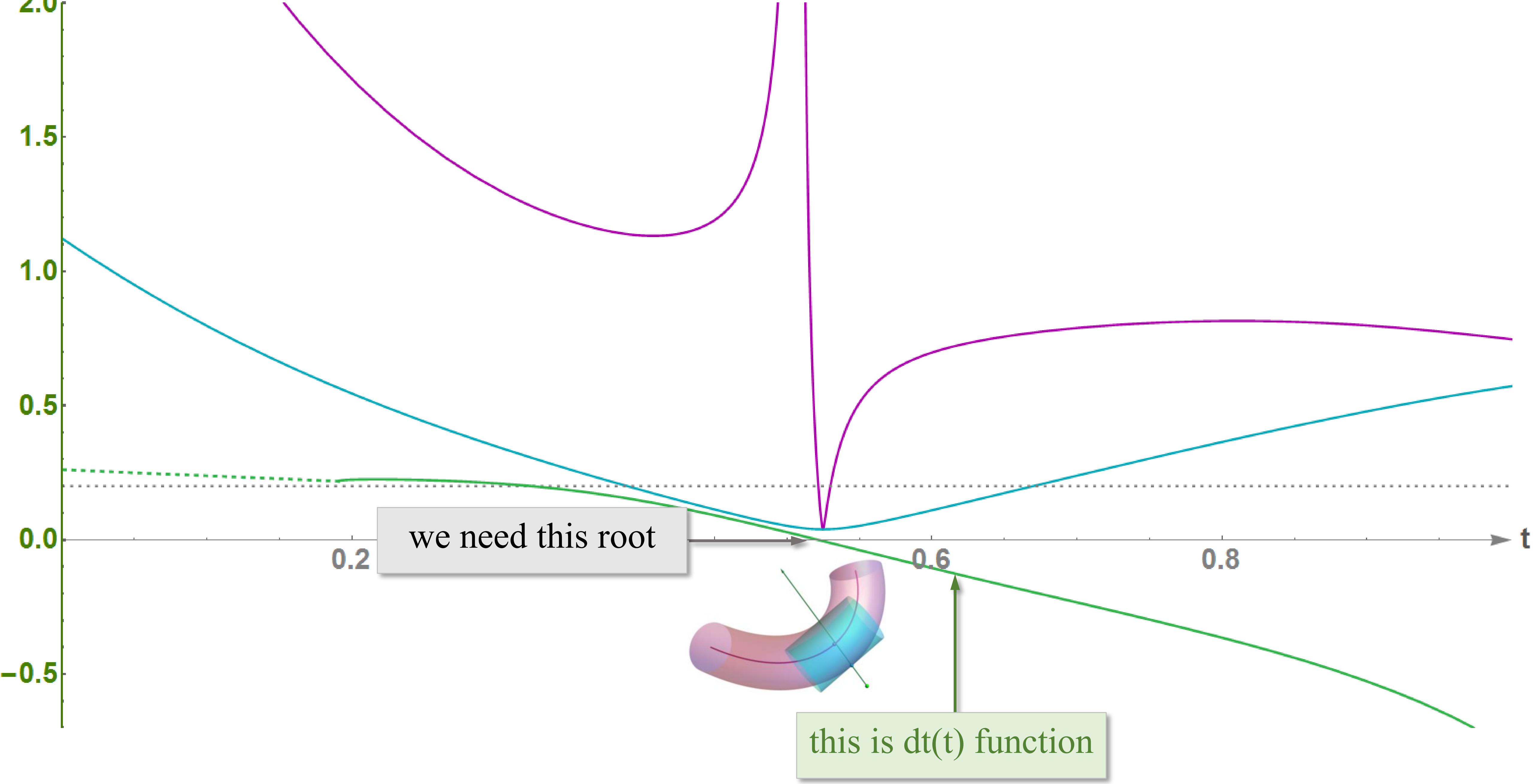


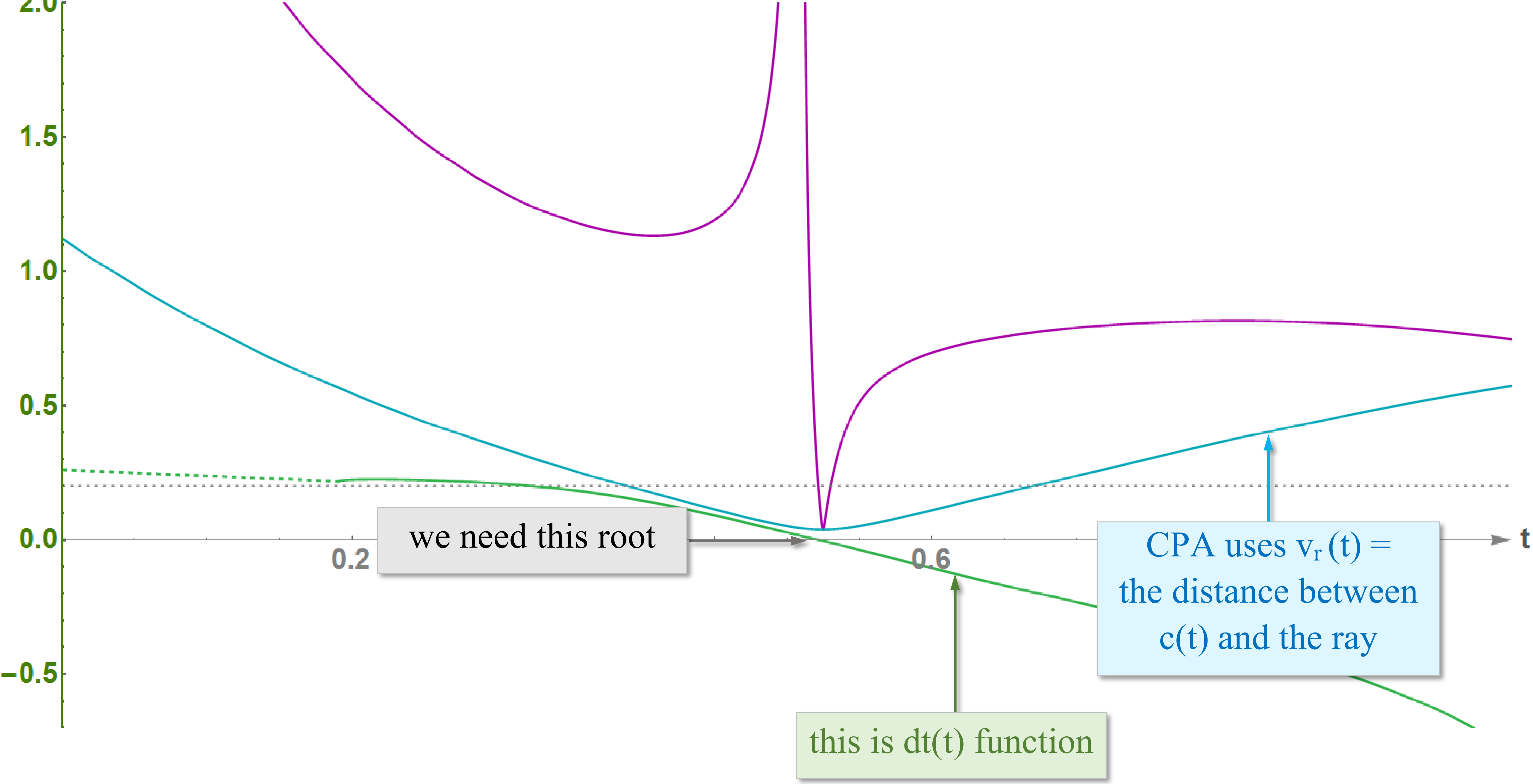
when dt(t) == 0,
ray intersects
the cone and
the swept volume
@ the same point ●

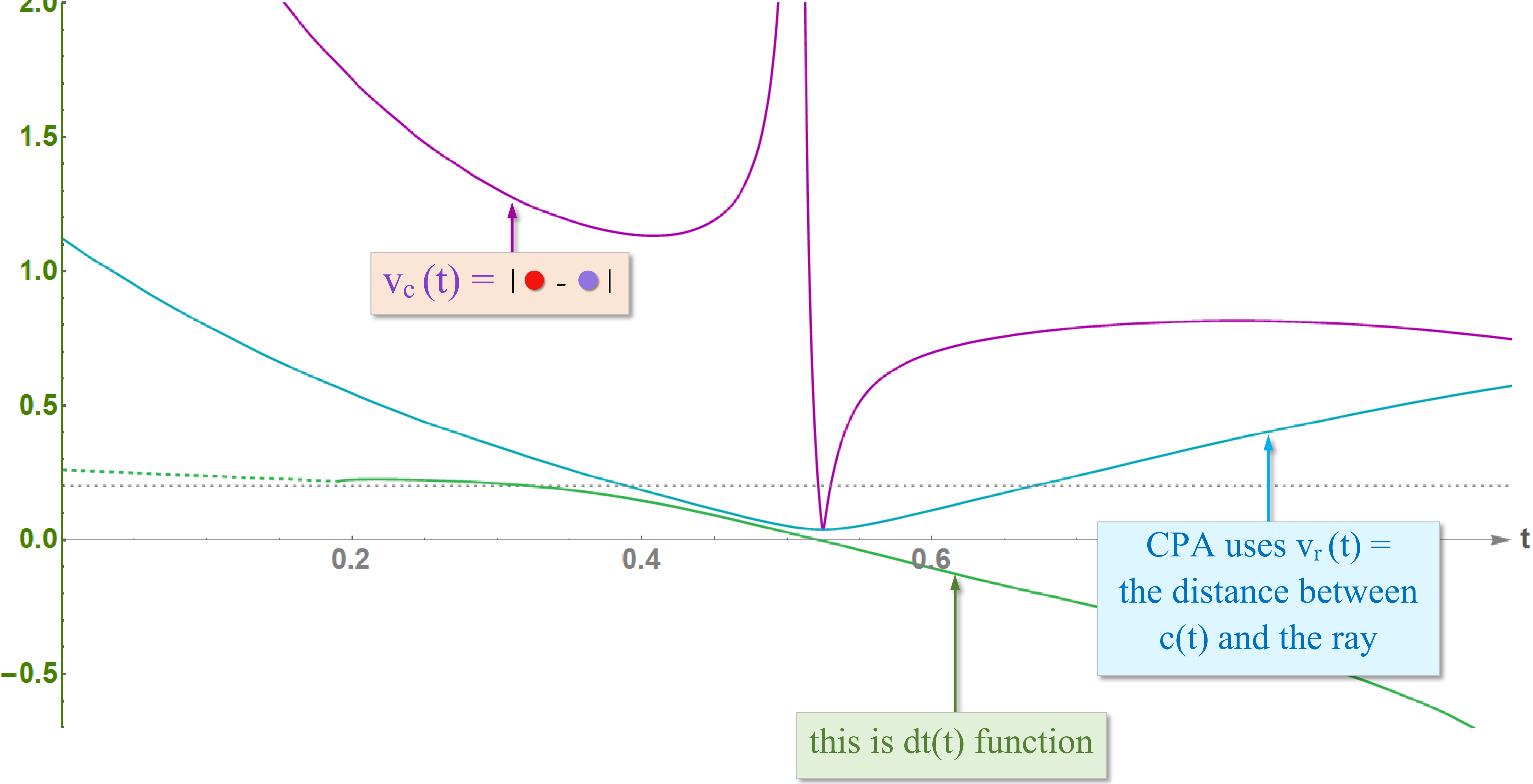


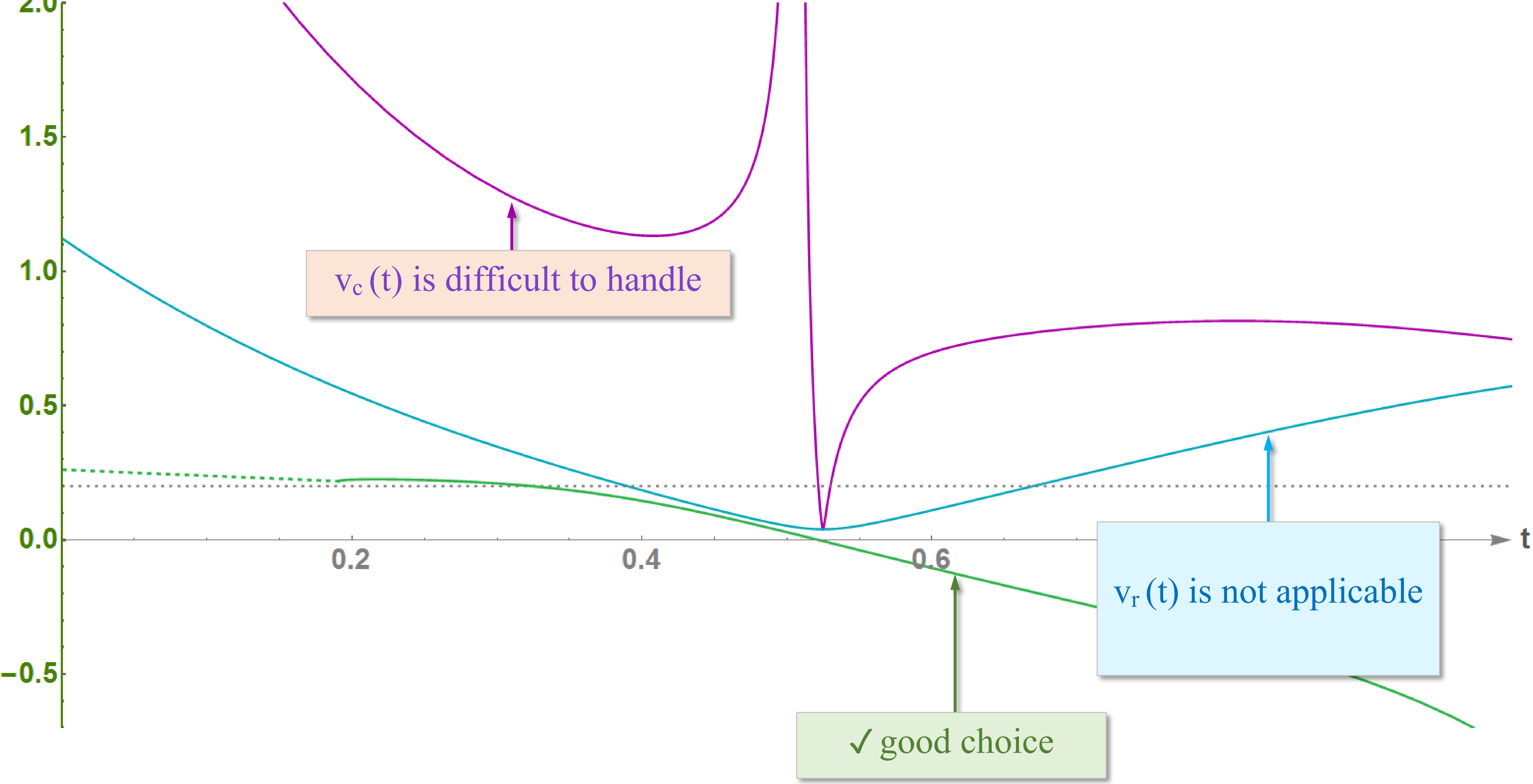
only the padded cone is intersected

this is dt(t) function

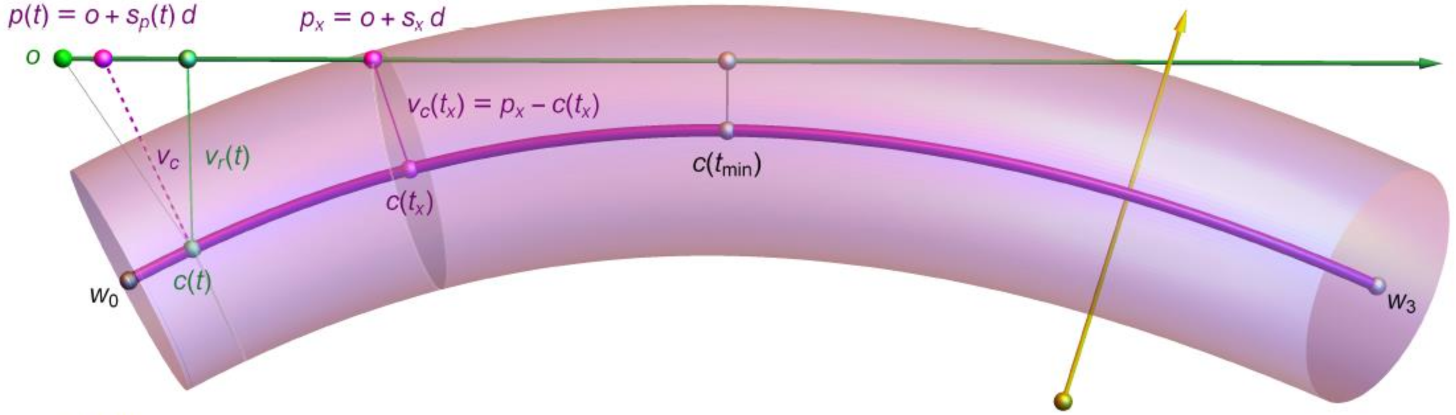

1.5
1.0
0.5
0.0
−0.5
0.2
0.6
0.8
t
we need this root
this is dt(t) function


2.0
1.5
1.0
0.5
0.0
−0.5
0.2
0.6
t
we need this root
this is dt(t) function
CPA uses $v_r(t)$ = the distance between c(t) and the ray


1.5
1.0
0.5
0.0
−0.5
0.2
0.4
0.6
t
$v_c(t) = |$ ● - ● $|$
CPA uses $v_r(t)$ = the distance between c(t) and the ray
this is dt(t) function

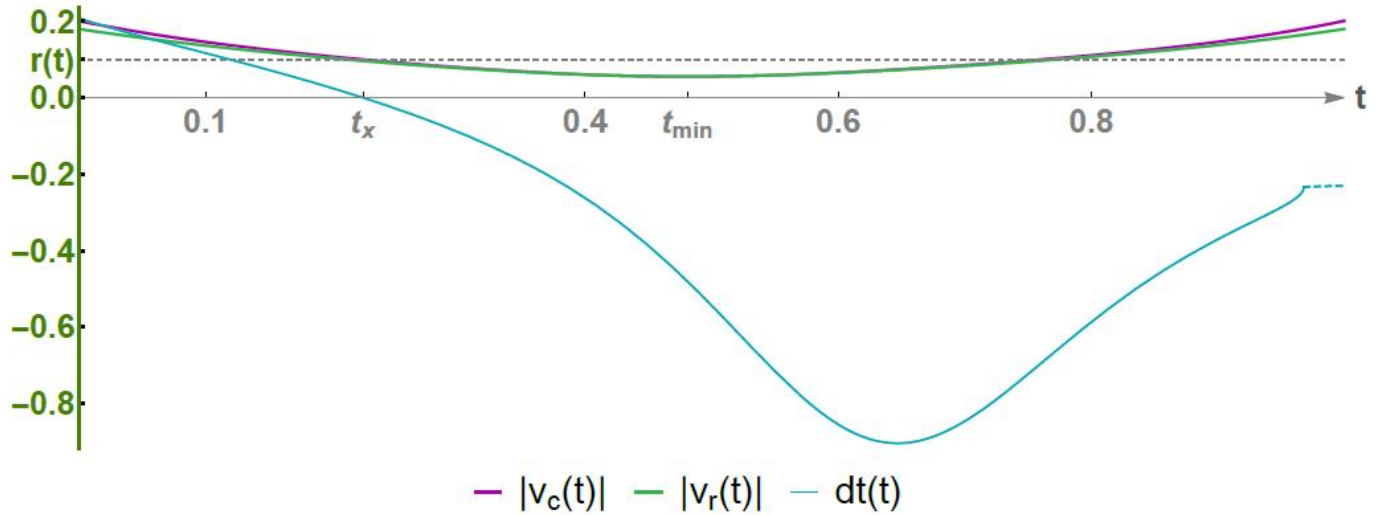


$v_c\,(t)$ is difficult to handle
$v_r\,(t)$ is not applicable
✓ good choice
1.5
1.0
0.5
0.0
−0.5
0.2
0.4
0.6
t

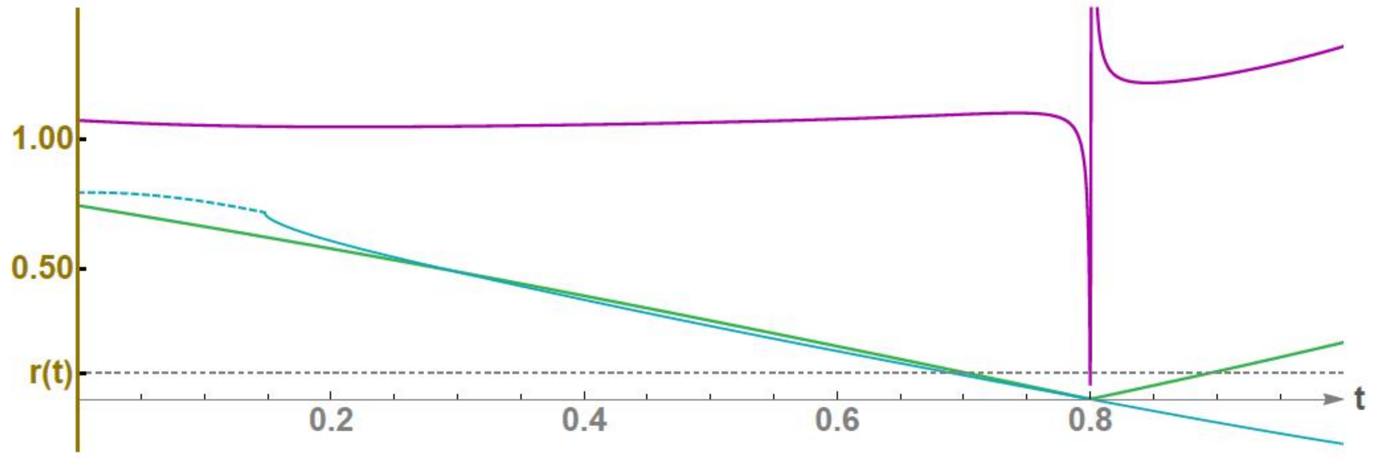


ray || curve

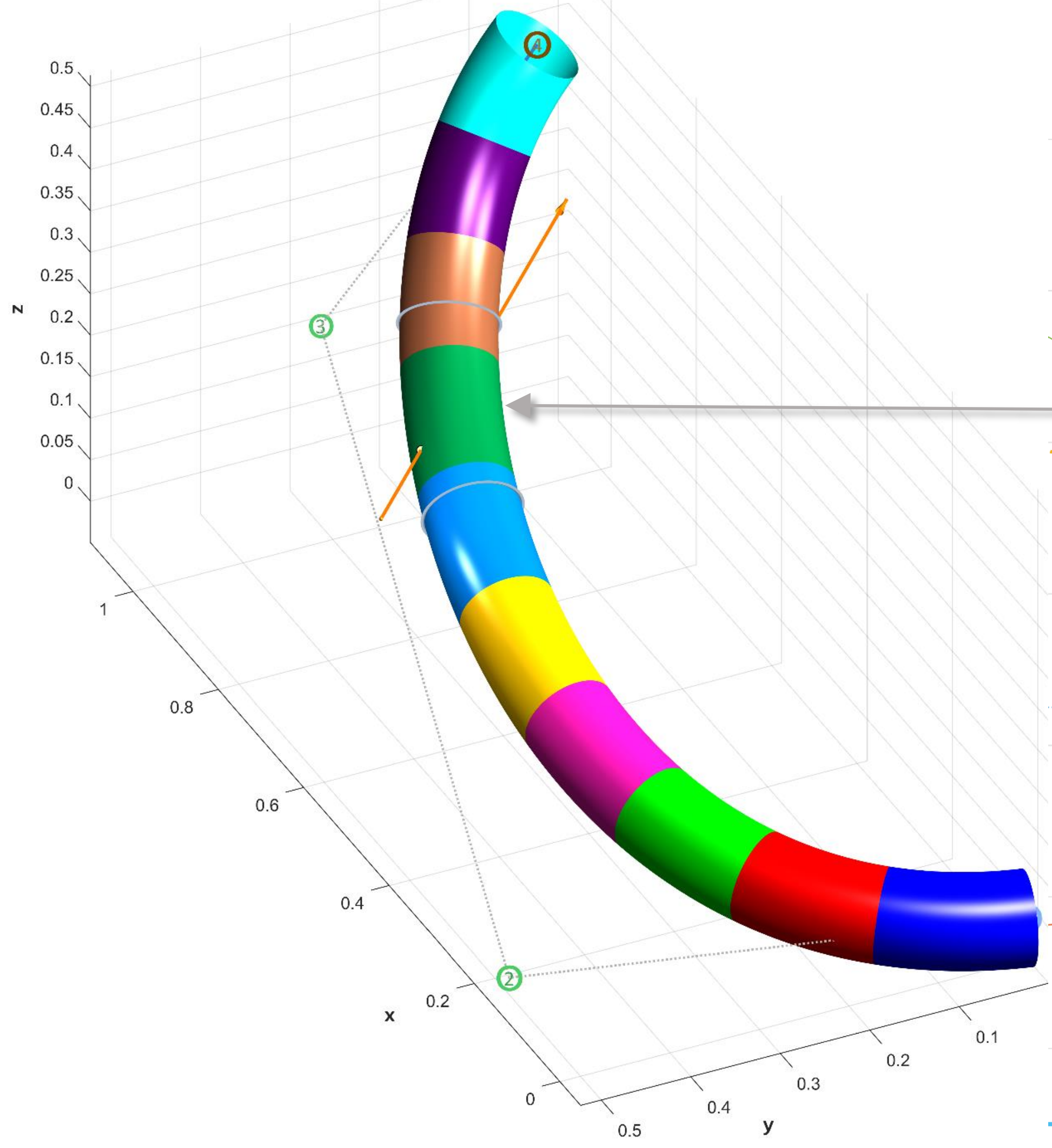
(more or less)

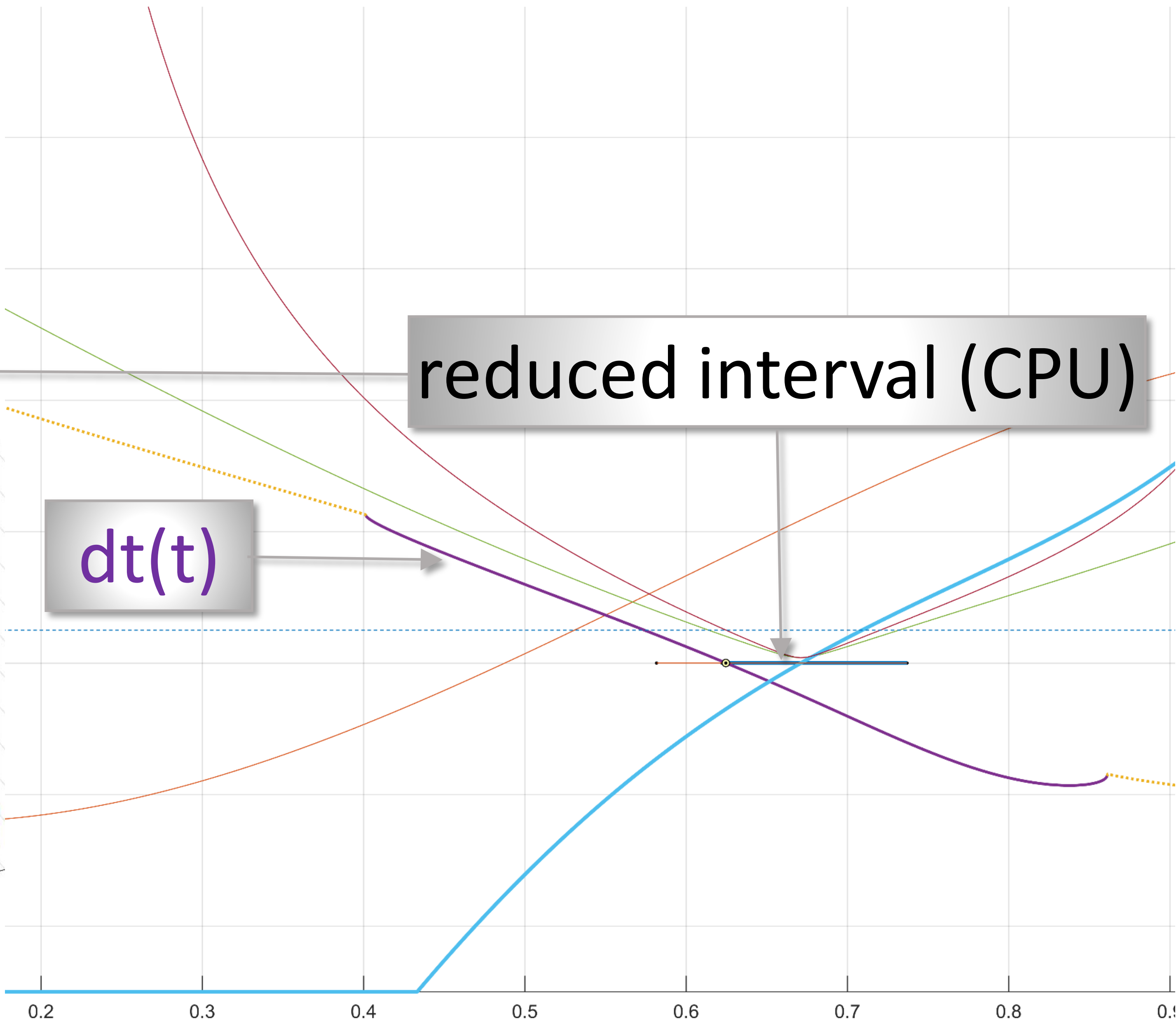


ray ⊥ curve

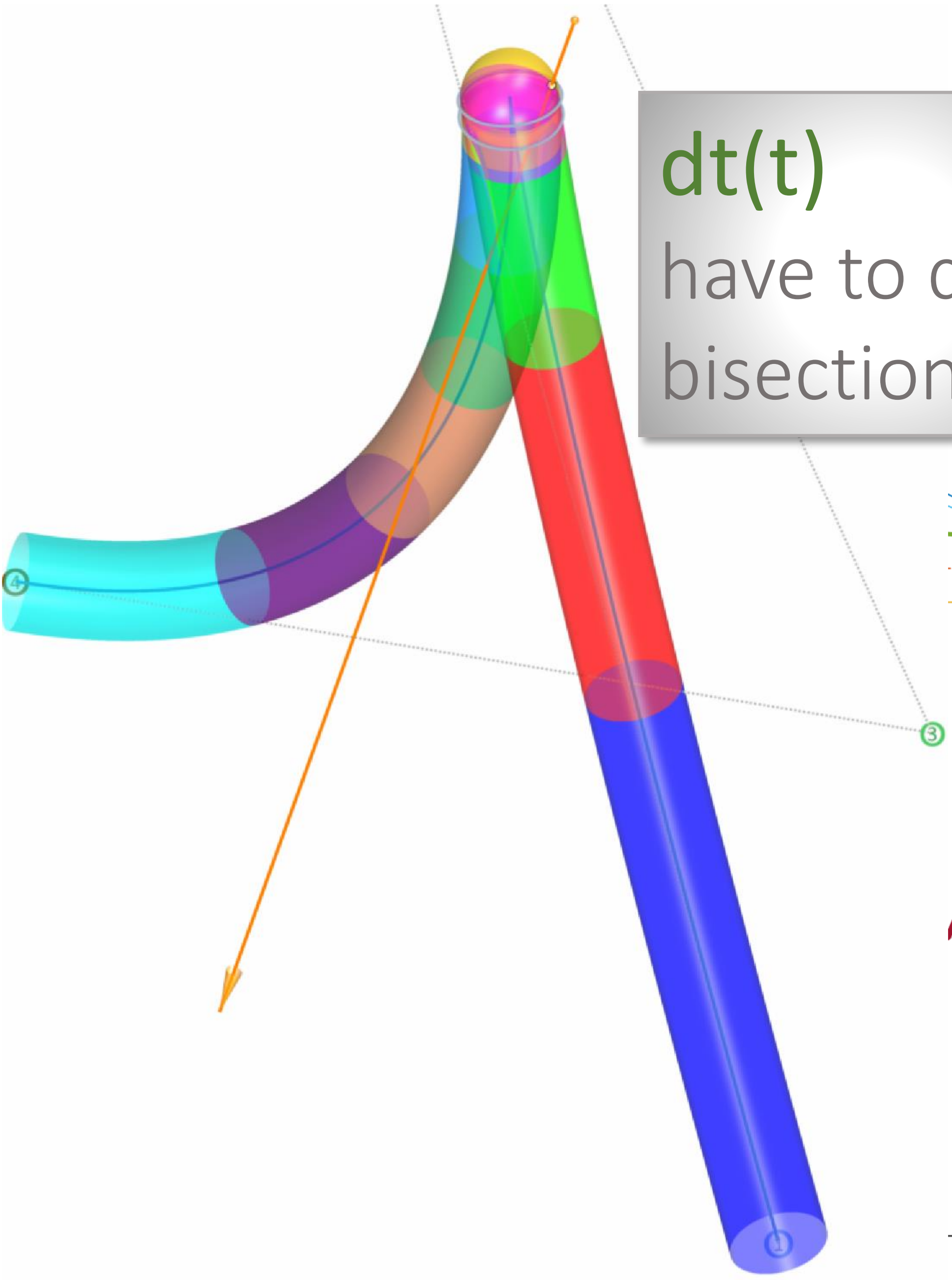


Typical situation: 3 iterations with *regula falsi*



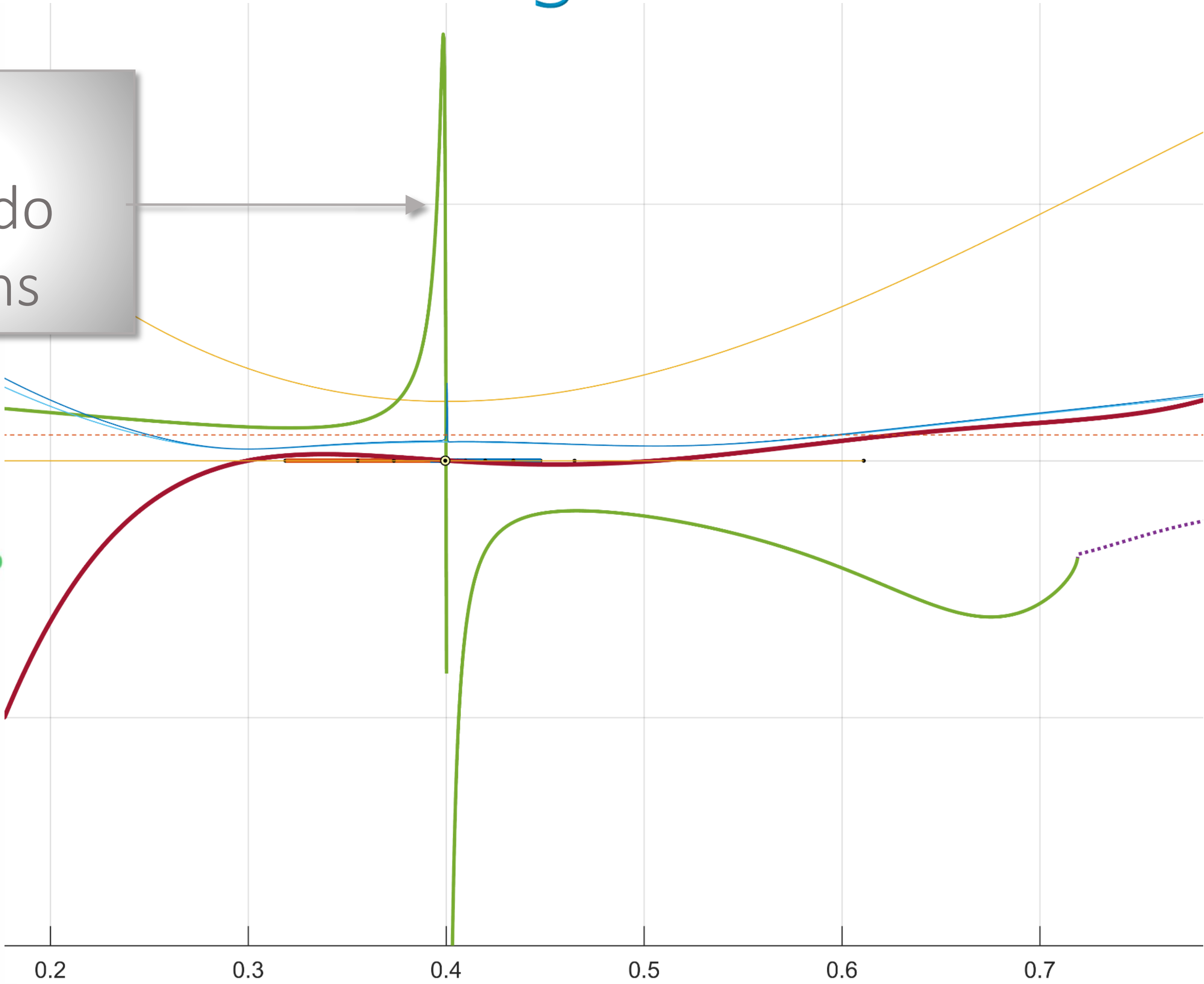


Difficult case for finding dt roots



dt(t)

have to do bisections



Details

Steps of the Phantom Intersector

1. Find (the exact) bounding box for the Bézier curve + padding during BVH building.
2. Check the ray against the curve's enclosing cylinder.
   Exit if no such intersection exists.
3. Transform the curve into the ray-centric coordinate system.
4. Split big intervals.
5. For each subinterval $[t_1, t_2]$
   1. If $dt(t_1) < 0$ and $dt(t_2) > 0$, ignore the interval
   2. Start iterations at the endpoint that is closer to the ray origin
   3. Test for convergence. If the intersection is found, report it, otherwise start at the other endpoint.

Steps of the Phantom Intersector

1. Find (the exact) bounding box for the Bézier curve + padding during BVH building.
2. Check the ray against the curve’s enclosing cylinder.
   Exit if no such intersection exists.
3. Transform the curve into the ray-centric coordinate system.
4. Split big intervals.
5. For each subinterval $[t_1, t_2]$
    1. If $dt(t_1) < 0$ and $dt(t_2) > 0$, ignore the interval
    2. Start iterations at the endpoint that is closer to the ray origin
    3. Test for convergence. If the intersection is found, report it, otherwise start at the other endpoint.

Enclosures (usable in any intersector)

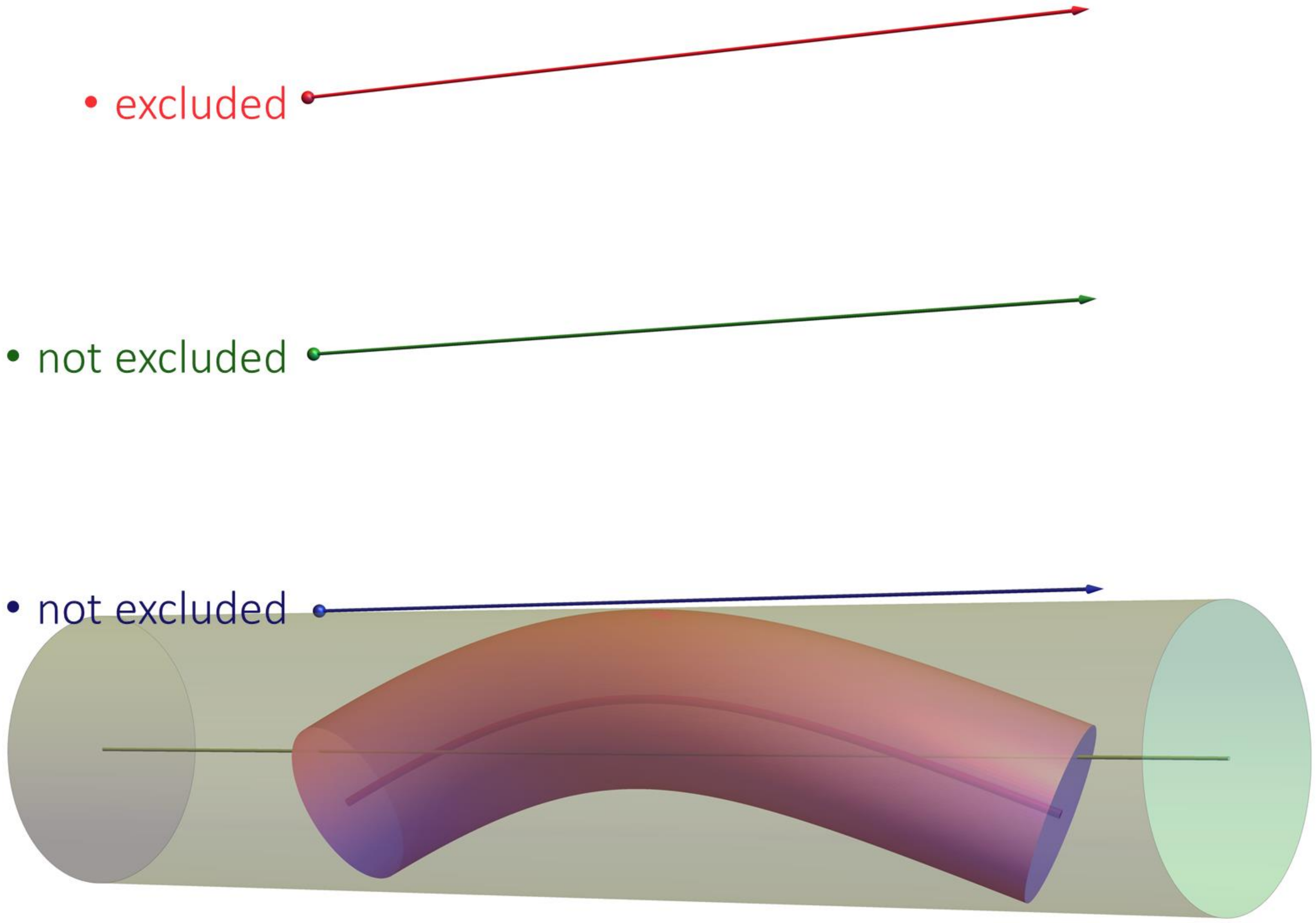
- While building the acceleration structure
  - (normalized) axis **acyl** = w_3 - w_0 (curve's endpoints)
  - point on the axis **pcyl** = ((w_3 + w_0)/2 + c(0.5))/2
  - find a conservative maximum distance **rcyl** from the points on the curve to this axis; add r_{max} to it and square
- Intersection is ruled out if the ray and the padded cylinder do not intersect (considered as infinite lines)

```
float3 dxc = cross(ray.direction, acyl);
float3 r2c = ray.origin - pcyl;
float  dl  = dot(r2c, dxc);
return dl * dl > rcyl * dot(dxc, dxc);
```

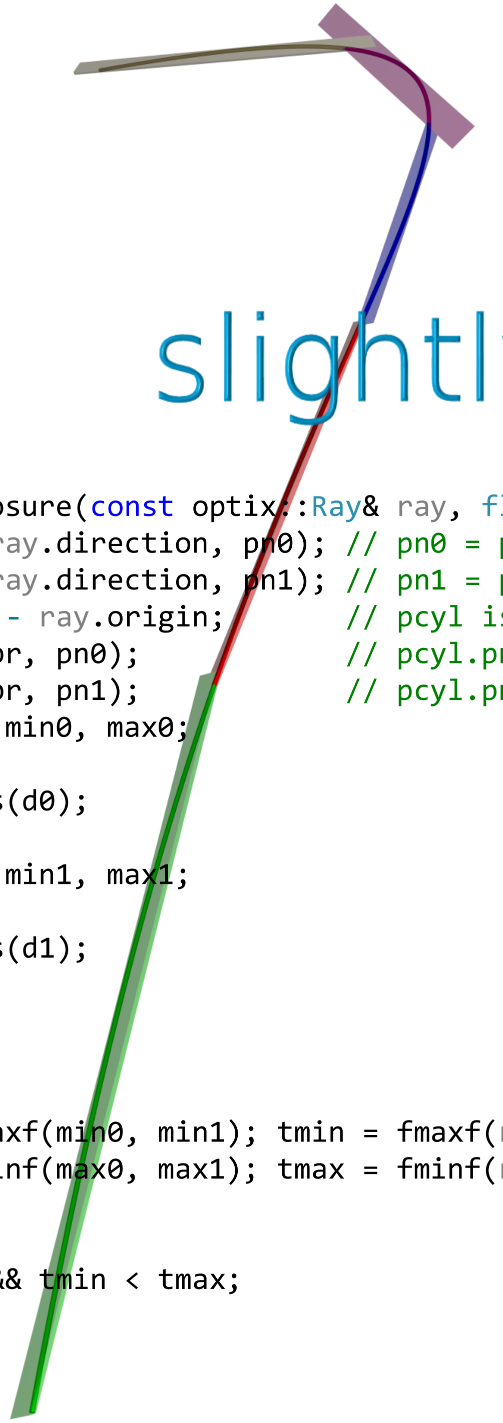
Enclosures (usable in any intersector)

- While building the acceleration structure
  - (normalized) axis **acyl** = $w_3 - w_0$ (curve's endpoints)
  - point on the axis **pcyl** = $((w_3 + w_0)/2 + c(0.5))/2$
  - find a conservative maximum distance **rcyl** from the points on the curve to this axis; add r_{max} to it and square
- Intersection is ruled out if the ray and the padded cylinder do not intersect (considered as infinite lines)

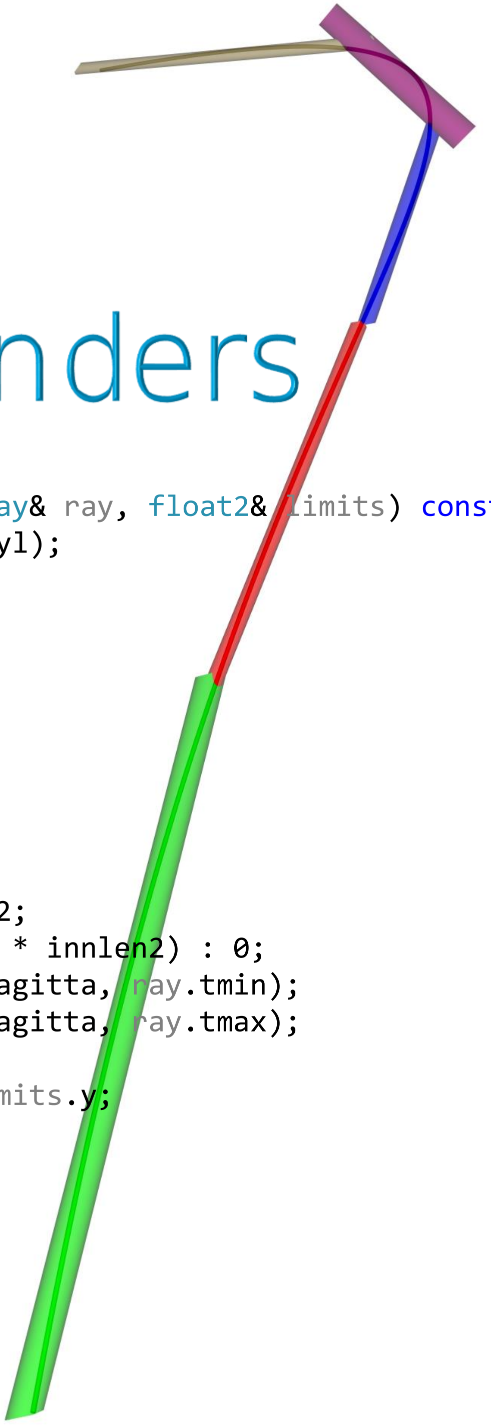
```
float3 dxc = cross(ray.direction, acyl);
float3 r2c = ray.origin - pcyl;
float  dl  = dot(r2c, dxc);
return dl * dl > rcyl * dot(dxc, dxc);
```


• excluded
• not excluded
• not excluded

slabs slightly slower than cylinders



```
bool intersectEnclosure(const optix::Ray& ray, float2& limits) const {
  float  d0 = dot(ray.direction, pn0); // pn0 = plane0.normal
  float  d1 = dot(ray.direction, pn1); // pn1 = plane1.normal
  float3 pr = pcyl - ray.origin;       // pcyl is a point on slab's axis
  float  p0 = dot(pr, pn0);            // pcyl.pn0 could be precomputed
  float  p1 = dot(pr, pn1);            // pcyl.pn1 could be precomputed
  float  pd0, ed0, min0, max0;
  pd0  = p0 / d0;
  ed0  = ext0 / abs(d0);
  min0 = pd0 - ed0;
  float  pd1, ed1, min1, max1;
  pd1  = p1 / d1;
  ed1  = ext1 / abs(d1);
  min1 = pd1 - ed1;
  max0 = pd0 + ed0;
  max1 = pd1 + ed1;

  float  tmin = fmaxf(min0, min1); tmin = fmaxf(ray.tmin, tmin);
  float  tmax = fminf(max0, max1); tmax = fminf(ray.tmax, tmax);
  limits.x = tmin;
  limits.y = tmax;
  return tmax > 0 && tmin < tmax;
}
```



```
bool intersectEnclosure(const optix::Ray& ray, float2& limits) const {
  float3 dxc = cross(ray.direction, acyl);
  float3 r2c = ray.origin - pcyl;
  float dl = dot(r2c, dxc);
  float innlen2 = 1.0f/dot(dxc, dxc);

  float d2 = dl * dl;
  float sagitta = rcyl - d2 * innlen2;

  float3 o2c = cross(r2c, acyl);
  float po2c = dot(o2c, dxc);
  limits.x = limits.y = -po2c * innlen2;
  sagitta  = sagitta > 0? sqrt(sagitta * innlen2) : 0;
  limits.x = optix::fmaxf(limits.x - sagitta, ray.tmin);
  limits.y = optix::fminf(limits.y + sagitta, ray.tmax);

  return limits.y > 0 && limits.x < limits.y;
}
```