# Rigid Bodies, Water Surface and Buoyancy

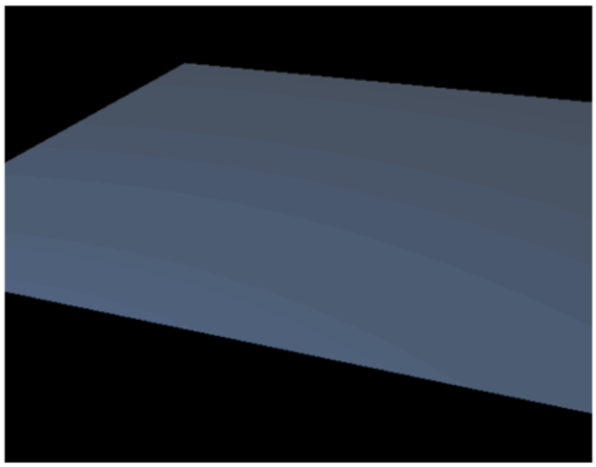
Randy Gaul - RandyGaul.net – r.gaul@digipen.edu

Special thanks to:
Erin Catto
Stan Melax
Bob Cromwell

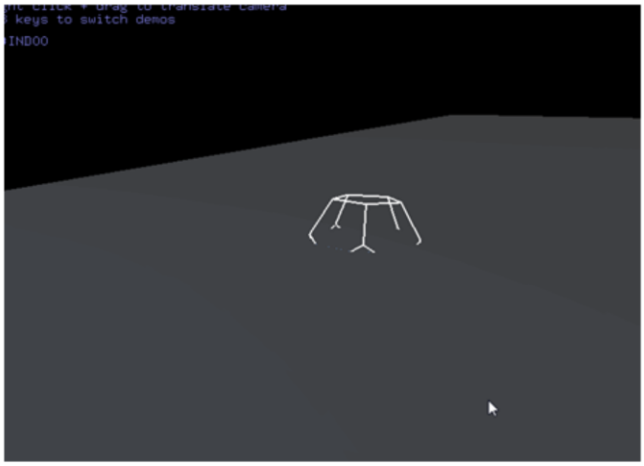
# Overview

- Buoyancy
- Triangle Clipping
- Surface Cloth
- Best-Fit Plane

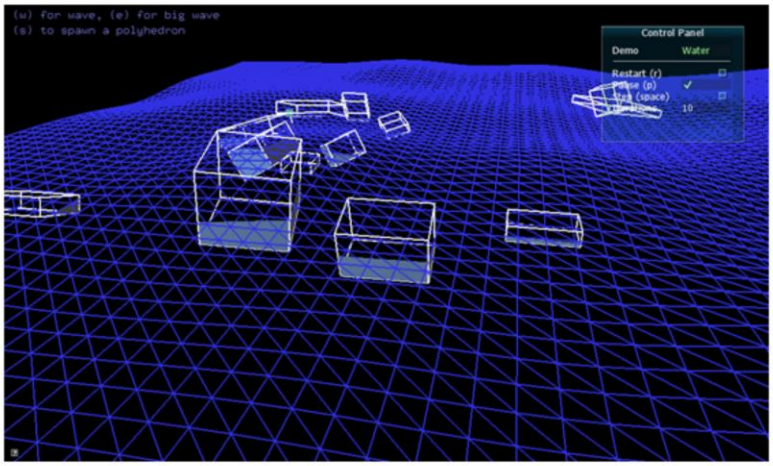
# Demo



# More! Demo



# More! Demo


(w) for wave, (e) for big wave
(s) to spawn a polyhedron
Control Panel
Demo
Water
Restart (r)
Pause (p)
(space)
10

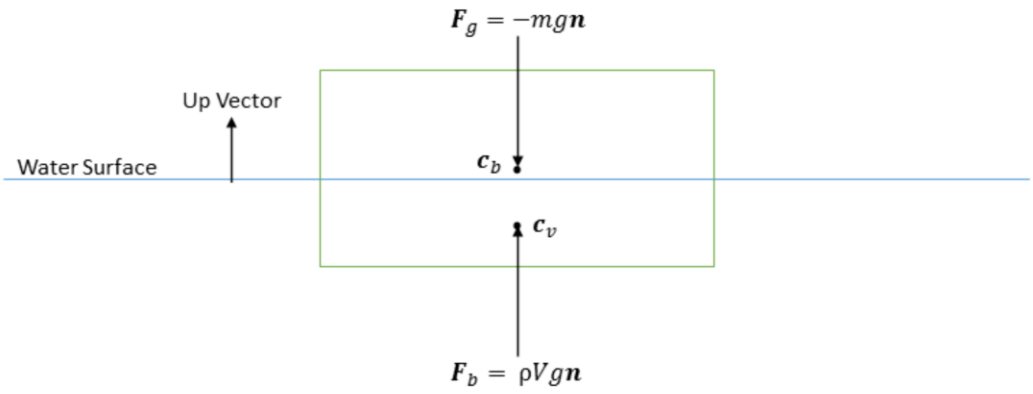
# Buoyancy

- Archimedes' principle
  - Buoyant force is equal to the weight of displaced fluid

$$\boldsymbol{F}_b = \rho V g \boldsymbol{n}$$

- ρ is "Rho"
  - Density of liquid
- V – volume of submerged portion of body
- g – gravity
- n – up vector (0, 1, 0)

# Buoyancy




Buoyancy counteracts gravity and can form an equilibrium. An equilibrium like in this diagram will result in oscillation of the polyhedron, so long as the two forces are not equal and opposite. This sort of oscillation is natural in nature, but diminishes quickly over time in the case of objects floating in water. Water exerts drag forces upon objects, of which will be approximated in later slides.

# Buoyancy

- $\boldsymbol{c}_b$ not always aligned with $\boldsymbol{c}_v$
  - Torque ($\boldsymbol{T}$) can be produced about $\boldsymbol{c}_b$

$$\boldsymbol{r} = \boldsymbol{c}_v - \boldsymbol{c}_b$$

$$\boldsymbol{T} = \boldsymbol{r} \times \rho V g \boldsymbol{n}$$

- $r$ is radius from center of body to center of submerged volume

# Buoyancy - Volume

- Volume of convex polyhedron required to satisfy:

$$\boldsymbol{F}_b = \rho V g \boldsymbol{n}$$

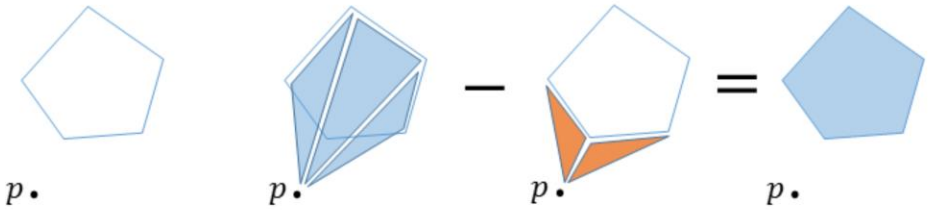
- Polyhedrons are made of triangles
- Each triangle can represent some volume

The only unknown required to apply buoyancy forces is the volume of the submerged portion of a given polyhedron. This must be solved in order to apply exact buoyancy and result in as realistic of a simulation as possible.

# Volume of Polygon

- Given reference point $p$ (use origin to simplify computation)
- For each edge $\{a, b\}$ form triangle $\{p, a, b\}$ with normal $\hat{\boldsymbol{n}}$
- Sum area of each triangle:

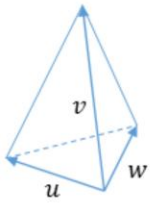
$$\boldsymbol{A} = \frac{1}{2}\sum (a - p) \times (b - a) \cdot \hat{\boldsymbol{n}}$$




It should be noted that the orange triangles represent “negative area”. The summation of both positive and negative values results in the desired area calculation. This will break down, numerically, of the polygon in question is too far from the origin. Polygons can be translated to the origin as a pre-step to increase numerical accuracy, if such a problem arises.

# Volume of Polyhedron

- Exact same idea except:
  - Form tetrahedrons from triangles and sum tetrahedron volumes



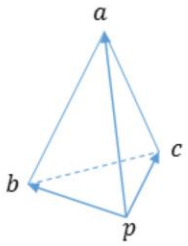


$$V = \frac{1}{6}\sum u \times v \cdot w$$

In order to simplify the calculation of volume greatly p should lay upon the water plane itself. This will be explained in a later slide's notes.

# Center of Mass in 3D

- Sum all tetrahedron centroids, weight them by volume
- Center of mass is called centroid $\boldsymbol{C}$




$$\boldsymbol{C}_i = \frac{1}{4}(a + b + c + p)$$

$$\boldsymbol{C} = \frac{1}{\boldsymbol{V}}\sum_{i=1}^{n} V_i C_i$$

# Volume and C.O.M of Polyhedron

- Centroid and volume and center of mass
  - Implicit origin reference

```
f32 TetVolume( Vec3& c, Vec3 u, Vec3 v, Vec3 w )
{
  real volume = (1.0f / 6.0f) * Dot( Cross( u, v ), w );
  c += (1.0f / 4.0f) * volume * (u + v + w);
  return volume;
}
```

Please note that this computation may not be optimal as the constant division operations can be factored out and applied once as an end-step. Since volume is being summed it may be very important to translate polyehdra vertices to the origin before computation occurs in order to avoid higher floating point granularity at distances farther from the origin. Error might be much worse (the summation value may reach great values) if constant division operations are factored out until the end. I have not done formal testing here, but these are all ideas I was aware of when writing my own implementation.

# Submerged Volume

- To compute $\boldsymbol{F}_b$ we need submerged volume:

- Clip all mesh triangles against water plane
- Use leftover triangles to compute volume

# Interior Triangle Clipping Routine

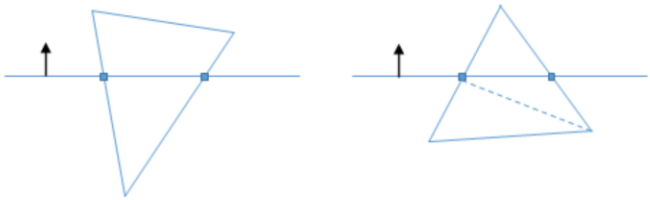
- Given triangle { $a$, $b$, $c$ } and plane $\boldsymbol{P}$
- Assume edge { $a$, $b$ } crosses clipping plane
- Compute distance $d$ of $c$ to $\boldsymbol{P}$
- Given $d$ orientation of $a$, $b$, and $c$ relative to $\boldsymbol{P}$ are now known
- Compute clipping results

The triangles formed by the "open hole" as a result of the clipping operation need not be considered so long as P lay upon the clipping plane.

If P is on the clipping plane then all triangles formed between the open ring edges in the clipped polyhedra and P will result in a volume of zero during the volume summation. It is important to pick a P reference point on the clipping plane itself in order to greatly simplify volume summation.

# Interior Triangle Clipping Routine

- Only two cases to consider
  - One vertex below plane, $a$ or $b$
    - One output triangle
  - Two vertices below plane, $a$ and $c$ or $c$ and $b$
    - Two output triangles



2 cases of clipping occur, but vertex A and B can be above and below the plane which results in 4 total cases in-code.

# Interior Triangle Clipping Routine

```
// Compute intersection point of a line segment and plane
Vec2 Intersect( const Vec3& a, const Vec3& b, f32 da, f32 db )
{
  return a + (da / (da - db)) * (b - a);
}
```

One can think of this calculation as a linear interpolation from a to b, where the interpolant is the time t between a and b to the clipping plane.

Another way to think of this is to calculate a one dimensional barycentric coordinate from a to b, where the coordinate lay upon the clipping plane.

# Interior Triangle Clipping Routine

```
f32 ClipTri( Vec3& C, Vec3 a, b, c, f32 d1, d2, d3 ) {
  Vec3 ab = Intersect( a, b, d1, d2 );
  f32 volume = 0.0f;
  if(d1 < 0.0f) // b to c crosses
    if(d3 < 0.0f)
      Vec3 bc = Intersect( b, c, d2, d3 );
      volume += TetVolume( C, bc, c, a );
      volume += TetVolume( C, ab, bc, a );
    else // c to a crosses
      Vec3 ac = Intersect( a, c, d1, d3 );
      volume += TetVolume( C, a, ab, ac );
  else // c to a crosses
    if(d3 < 0.0f)
      Vec3 ac = Intersect( a, c, d1, d3 );
      volume += TetVolume( C, ac, ab, b );
      volume += TetVolume( C, b, c, ac );
    else // b to c crosses
      Vec3 ac = Intersect( b, c, d2, d3 );
      volume += TetVolume( C, b, bc, ab );
  return volume;
}
```

Assuming a and b cross the clipping plane (either a above and b below, or a below and b above), all that needs to be found is whether c lay above or below the clipping plane. This results in 4 different cases.

Since it is known that a and b (or rather, assumed) are on opposite sides of the clipping plane once the orientation of a is known, b is implicitly known as well. This reduces the total number of if statements and adds to code-compactness.

# Exterior Triangle Clipping Routine

- Earlier assumption:
  - edge $\{ a, b \}$ crosses clipping plane
- First, find *any* edge that crosses clipping plane

It is important to realize that the assumption we made earlier about a and b laying upon opposite sides of the clipping plane is an invariant that must be supplied manually. This means we need to find two vertices to *act* as a and b.

# Find *Any* Edge { $a$, $b$ }

- Compute distance of each vertex to water plane

```
// Plane form:
//  n - Unit normal of the plane
//  d - Scalar distance of plane
//      to origin

f32 d1 = Dot( a, n ) - d;
f32 d2 = Dot( b, n ) - d;
f32 d3 = Dot( c, n ) - d;
```

# Find *Any* Edge { $a, b$ }

- Detect if any edge on { $a, b, c$ } crosses water surface

```
// a to b crosses the clipping plane
if(d1 * d2 < 0.0f)
  clipVolume = ClipTriangle( C, a, b, c, d1, d2, d3 );

// a to c crosses the clipping plane
else if(d1 * d3 < 0.0f)
  clipVolume = ClipTriangle( C, c, a, b, d3, d1, d2 );

// b to c crosses the clipping plane
else if(d2 * d3 < 0.0f)
  clipVolume = ClipTriangle( C, b, c, a, d2, d3, d1 );

// Full clipping plane intersection; keep the whole triangle
else if(d1 < 0.0f || d2 < 0.0f || d3 < 0.0f)
  clipVolume = TetVolume( C, a, b, c );
```

Once *any* edge is found to have vertices on opposite sides of the clipping plane, this edge can immediately be referred to as { a, b }. The triangle is re-ordered and passed into the interior clipping routine covered in previous slides, where we assumed a and b crossed the clipping plane.

The last case is for when the whole triangle is submerged.

# Drag Force

- Buoyant force and gravity coupled = oscillation
- Drag forces diminish oscillation over time
  - Drag forces can also let current move objects
- Realistic drag simulation requires full water simulation
  - Erin Catto provides the following drag approximation formulae

# Catto Drag Force Approximations

- Linear force of drag ($\boldsymbol{F}_d$) approximation

$$\boldsymbol{F}_d = \beta_l m \frac{V_l}{V_p}(v_l - v_b)$$

- $\beta_l$ – linear drag coefficient in Hz
- $V_l$ – volume of the submerged liquid
- $V_p$ – volume of polyhedron
- $v_l$ – velocity of liquid
- $v_b$ – velocity at center of submerged volume from rigid body

# Catto Drag Force Approximations

- Angular drag torque ($\boldsymbol{T}_d$) approximation

$$\boldsymbol{T}_d = \beta_a m \frac{V}{V_b} L^2 \boldsymbol{\omega}$$

- $\beta_a$ – angular drag coefficient in Hz
- $L$ – approximate length of polyhedron
- $\boldsymbol{\omega}$ – angular velocity of rigid body

L can be calculated in-code as the radius from the center of the polyhedron to the center of the submerged volume. This radius length can be cached and reused later, and the longest recorded radius can be used to approximate the length of the polyhedron in question. Actual length of the radius vector need not be calculated, and instead a more optimal version (computationally speaking) of length squared can be calculated and directly substituted into the above equation.

# Simulating the Water Surface

- Lets make this easy and super fast
  - Not physically based
- No real math involved
- Model water surface as a simple cloth

# Simulating the Water Surface

- 2D array of heights (representing point masses)

```
struct WaterParticle
{
  real m_height;
  real m_oldHeight;
  real m_targetHeight;
  real m_velocity;
};
```

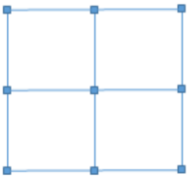
- Can calculate x and y position implicitly from grid
  - Lower memory footprint and increase cache coherency

Target height can be a global value to further reduce memory footprint. Memory footprint is great to minimize in an array like this since we're running this code on cache-hungry CPU hardware.

# Simulating the Water Surface

- Connect each particle to all adjacent neighbors
  - No need for shear or bend edges!

```
struct WaterEdge
{
  uint32 a;
  uint32 b;
};
```



Shear and bend edges are common in cloth simulation, but since our particle grid only acts as a height array the particles themselves cannot actually shear or bend (by definition) as a result.

# Simulating the Water Surface

- Store faces for rendering (optional)
  - Can implicitly compute these too

```
struct WaterFace
{
  uint32 a;
  uint32 b;
  uint32 c;
};
```

I myself use an array of faces to simplify implementation ☺

# Simulating the Water Surface

- Assume water surface is in some non-settled configuration
- How do we solve the cloth to enforce proper particle behavior?

```
void WaterSurface::Solve( f32 dt )
{
  // ... ?
}
```

# Simulating the Water Surface

- Lets pull particles towards a desired surface height

```
void WaterSurface::SolveDepths( void )
{
  for(uint32 i = 0; i < m_particleCount; ++i)
  {
    WaterParticle *p = m_particles + i;

    real x = p->m_targetHeight - p->m_height;
    p->m_height += x * m_stiffness;
  }
```

- Stiffness – unit-less interpolant from 0 to 1

Stiffness is just a parameter to describe how much the particles try to conserve total volume of the water. It is a unit-less tuning parameter. Various values for stiffness can result in a wide range of surface behavior.

# Simulating the Water Surface

- Now we have a jiggly point-mass array
- Lets pull particles towards neighbor heights

```
void WaterSurface::SolveEdges( void ) {
  for(uint32 i = 0; i < m_edgeCount; ++i) {
    WaterEdge *e = m_edges + i;
    WaterParticle *a = m_particles + e->a;
    WaterParticle *b = m_particles + e->b;

    real d = b->m_height - a->m_height;
    d *= m_propogation; // another interpolant

    a->m_height += d;
    b->m_height -= d;
  }
}
```

Propagation is another unit-less tuning parameter, and just describes how much particles are pulled to their neighbor's height. How many times we run the SolveEdges function within a given simulation loop determines how many edges a wave can propagate across the particle grid.

This results in two more tuning parameters: propagation value and edge solve iterations. These two parameters can result in a wide range of behaviors. Obviously higher SolveEdges iteration counts will slow down performance.

# Simulating the Water Surface

- Main solve loop (some black boxes for now)

```
void WaterSurface::Solve( f32 dt )
{
  Integrate( dt );

  for(uint32 i = 0; i < 5; ++i)
    SolveEdges( );

  SolveDepths( );

  VelocityFixup( 1.0f / dt );
}
```

# Simulating the Water Surface

- Integration
  - Okay, a little real math is used (I lied)

```
void WaterSurface::Integrate( f32 dt )
{
  for(uint32 i = 0; i < m_particleCount; ++i)
  {
    WaterParticle *p = m_particles + i;

    p->m_height += dt * p->m_velocity;
  }
}
```

# Simulating the Water Surface

- Velocity fixup
  - Since we modify positions directly this is necessary

```
void WaterSurface::VelocityFixup( f32 inv_dt )
{
  for(uint32 i = 0; i < m_particleCount; ++i)
  {
    WaterParticle *p = m_particles + i;

    real delta = p->m_height - p->m_oldHeight;
    p->m_velocity = inv_dt * delta;
    p->m_oldHeight = p->m_height;
  }
}
```

# Clipping Planes

- WAIT A MINUTE
- Waves have a curved surface
  - Our clipping only handles a flat plane

# Clipping Planes

- Possible approximate solution:

- Project polyhedron into 2D water frame
- Generate a 2D bounding volume
  - AABB, circle, anything simple will work
- Find all intersecting point-mass (ignore heights)
- Assign intersecting points to set $\boldsymbol{S}$
- Compute best fit plane of $\boldsymbol{S}$

# Best Fit Plane from $\boldsymbol{S}$

- Given point set $\boldsymbol{S}$ compute best fit plane:

- Compute covariance matrix $\boldsymbol{C}$ of $\boldsymbol{S}$
- Calculate eigenvalues $\{ e_1, e_2, e_3 \}$ of $\boldsymbol{C}$
- Calculate eigenvectors associated with each eigenvalue

# Best Fit Plane from $\boldsymbol{S}$

- Compute covariance matrix $\boldsymbol{C}$ of $\boldsymbol{S}$

- Covariance is the sum of all outer products of all elements of $\boldsymbol{S}$ with themselves
- Outer product of two vectors returns a matrix

$$\boldsymbol{C} = \sum_{i=1}^{n} v_i \otimes v_i \qquad v \otimes v = \begin{bmatrix} {v_x}^2 & v_x v_y & v_x v_z \\ v_y v_x & {v_y}^2 & v_y v_z \\ v_z v_x & v_z v_y & {v_z}^2 \end{bmatrix}$$

# Best Fit Plane from $\boldsymbol{S}$

- Calculate eigenvalues $\{ e_1, e_2, e_3 \}$ of $\boldsymbol{C}$
- Eigenvalue equation for a covariance matrix $\boldsymbol{A}$

$$\boldsymbol{A}v - \lambda v = 0$$

- Rearrange to find characteristic polynomial
  - Roots of cubic polynomial are the eigenvalues

$$\det(\boldsymbol{A} - \lambda I) = 0 = \lambda^3 + a\lambda^2 + b\lambda + c$$

There are iterative methods such as HouseHolder-QR/QL, Jacobi et al. that can compute an eigensolution (eigenvalues and corresponding eigenvectors) in an iterative fashion. These algorithms are useful in that they are generalized for n by n matrices, but closed form computation (as proposed in this slide) is much faster. Sadly closed-form solutions for polynomials for degree 5 and higher do not have formulas, which is why iterative methods have been studied for so long.

# Best Fit Plane from $\boldsymbol{S}$

- Solving cubic polynomial roots

- Cardano's method (Schwarze, Graphics Gems I pg. 404)
- Optimized method (Cromwell, Graphics Gems IV pg. 193)
  - This is specific to exactly 3x3 covariance matrix eigenvalues!

# Best Fit Plane from $\boldsymbol{S}$

- Solving for eigenvectors
- Classic linear algebra system of equations

$$\boldsymbol{A}x = b$$

- $\boldsymbol{A}$ – Covariance matrix
- $x$ – eigenvector associated to an eigenvalue
- $b$ – vector containing an eigenvalue
  - The same eigenvalue is placed in all components of $b$

# Best Fit Plane from $\boldsymbol{S}$

$$\boldsymbol{A}x = b, \qquad x = \begin{bmatrix} ? \\ ? \end{bmatrix}$$

- How does one solve a system of linear equations?
- Gaussian elimination is simple and efficient

# Recap

- Project polyhedron into 2D water frame, find $\boldsymbol{S}$
- Compute best fit plane of $\boldsymbol{S}$
- Clip polyhedron to water plane
  - Find volume and centroid of submerged volume
- Compute buoyant and drag forces
- Apply forces to rigid body
- Solve the water surface

# Make Splashes

- When polyhedron first hits the plane, make a splash!

- Idea:
- Hit water surface with a sphere
  - Points near sphere center feel large force
  - Points near sphere edge feel small force

# Make Splashes

```
void WaterSurface::MakeSplash( const Vec3& p, f32 r, f32 force )
{
  f32 r2 = radius * radius;
  for(uint32 i = 0; i < m_particleCount; ++i)
  {
    WaterParticle *p = m_particles + i;

    Vec3 pos = GetParticlePosition( i );
    real l = (pos - p).LengthSquared( );
    if(l < r2)
    {
      l = std::sqrt( l );
      real factor = (radius - l) / radius;
      p->m_oldHeight = p->m_height;
      p->m_height += force * factor;
    }
  }
}
```

# Limitations:

- Drag is approximated
  - Slightly unrealistic oscillation (bobbing)
- Clipping plane is approximated
  - Volume calculations are not clipped to exact cloth mesh
- Water surface is modelled as a fake cloth
  - No self-intersection
  - No cresting waves
  - Not physically based

# Pros:

- This is all really cool and really fast

# Questions?

# Resources

- Catto: Game Programming Gems 6 – Exact Buoyancy for Polyhedra
- Melax: GDC 2013 Math for Game Programmers – Interacting with 3D Geometry
- Cromwell: Graphics Gems IV pg. 193 – Cubic and Quartic Roots
- Schwarze: Graphics Gems I pg. 404 – Efficient Eigenvalues for Visualization