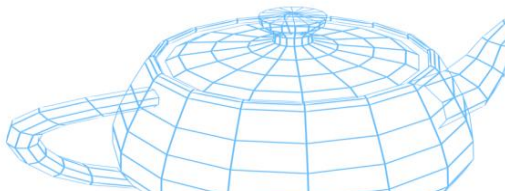
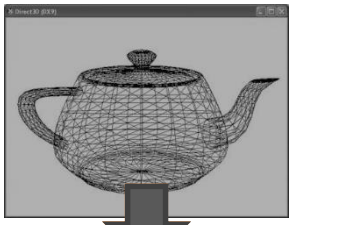
SUPSI

Computer Graphics

Transparency

Achille Peternier, lecturer





World Transformations

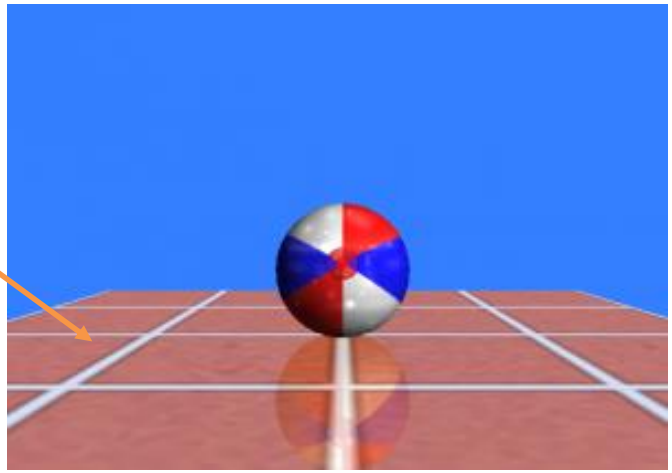
Lighting

Projection Transformations

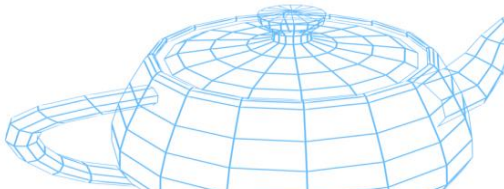
Clipping

Rasterization

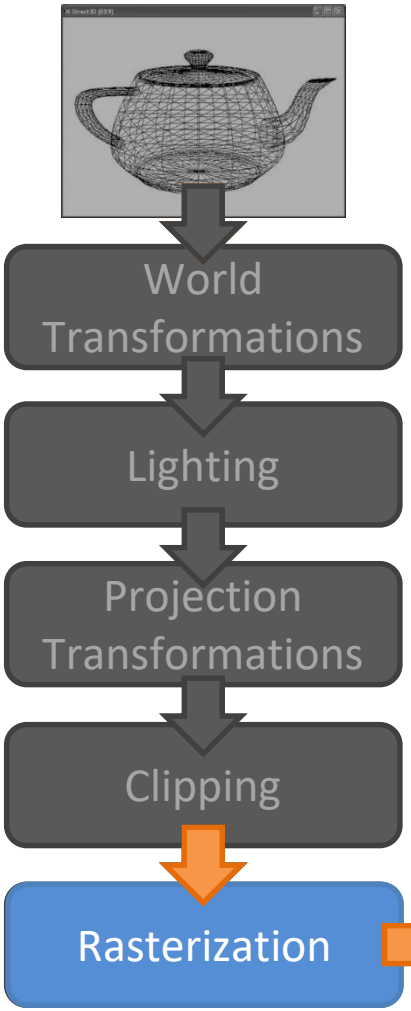
Transparent plane



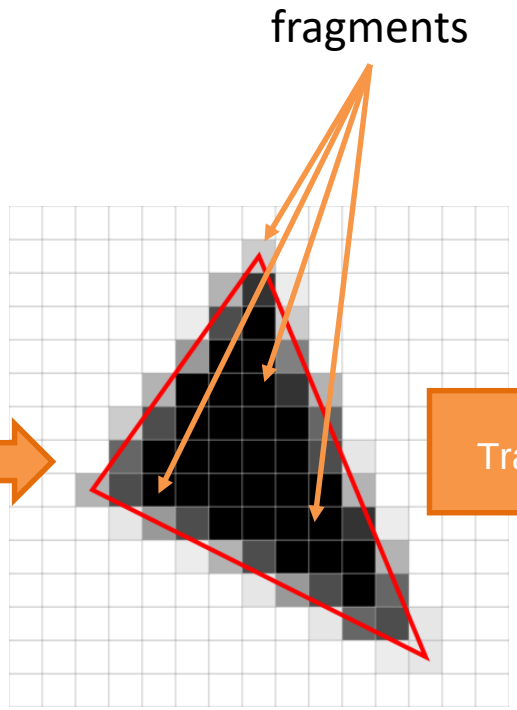
Transparency

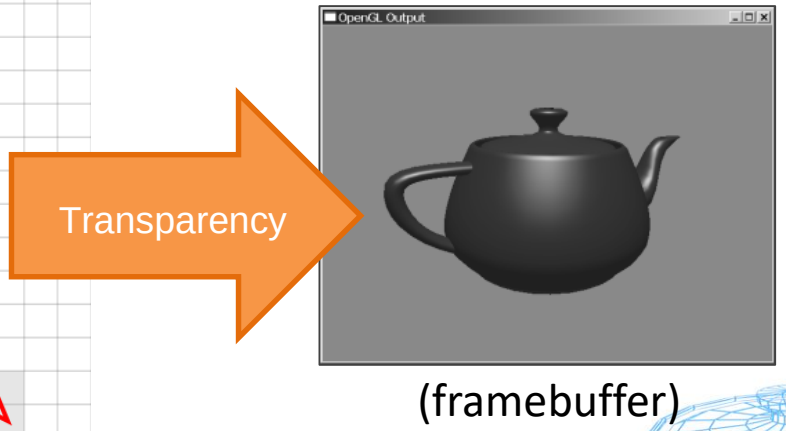


Fragments

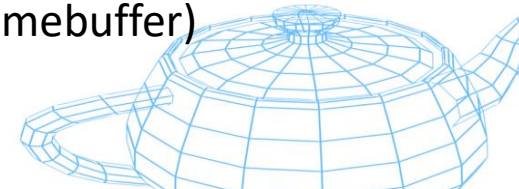


Fragments are the output of the rasterization step just before the writing to the framebuffer





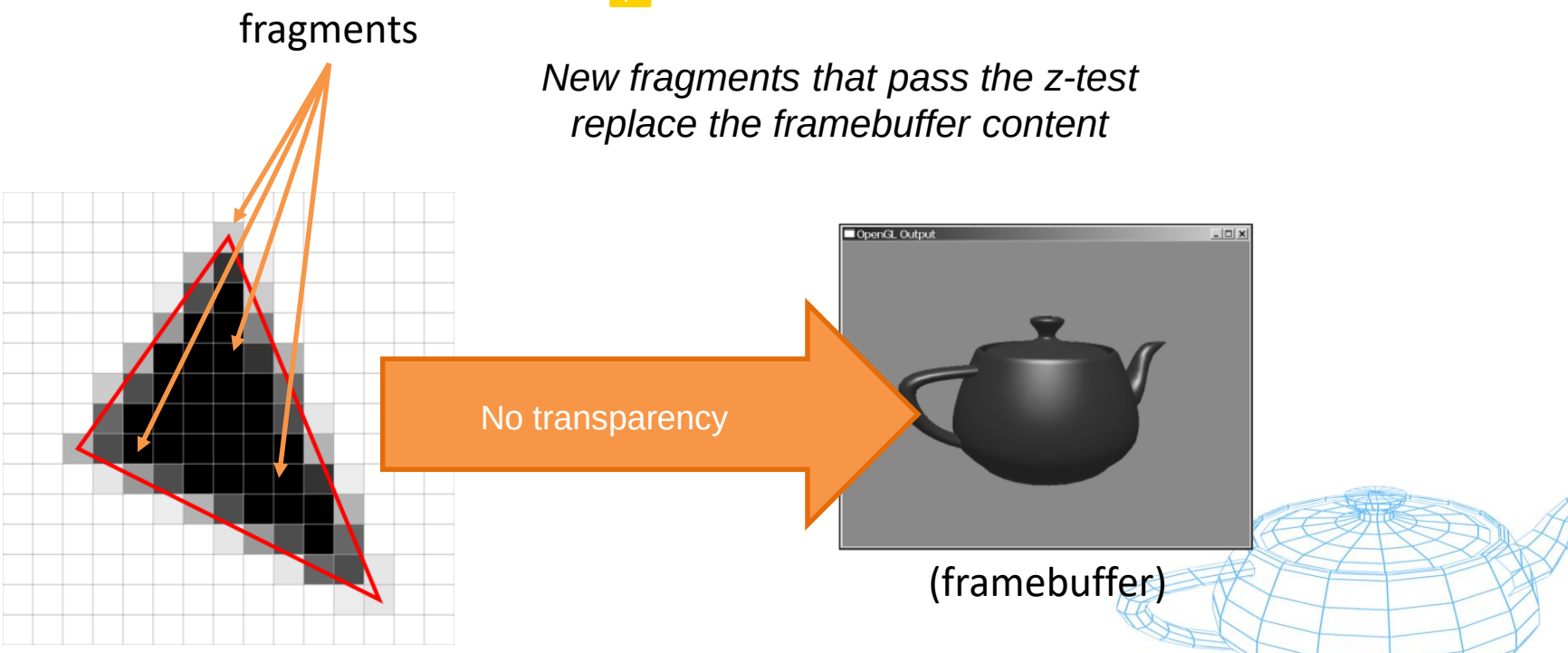
(framebuffer)



Without transparency

- When transparency is not used, output fragments generated during the rasterization are copied to the framebuffer (as long as they pass the depth test).

- Solid/opaque fragments simply replace the different color pixels already stored in the framebuffer.



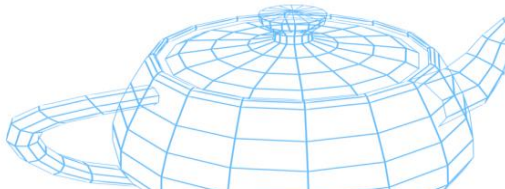
Transparency

- There are two main types of transparency: **hard** and **soft**.

- **Hard transparency** refers to a binary kind of transparency, where an element is either entirely solid (not transparent at all) or completely transparent (thus invisible):
  - Hard transparency is implemented in OpenGL through the **alpha test**.

- **Soft transparency** enables progressive transparency with mutable gradients of intensity to provide see-through effects:
  - Soft transparency is supported in OpenGL through **blending**.

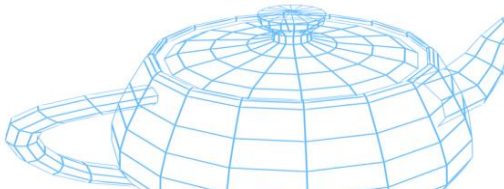
- Transparency intensity is commonly expressed through the alpha value (e.g., RGB**A**, `glColor4f()`, texture alpha channel, `glMaterial()` alpha component, etc.).



Transparency

- In OpenGL, transparency is applied in the **blending** step:
  - Blending is the last operation of the pipeline and consists in copying one fragment (as an element of a rendered primitive) to the framebuffer.
  - When transparency is used, instead of simply overwriting pixels in the framebuffer with new content, fragments are merged (added, multiplied, interpolated, etc.) with the pixels already there.

- Blending is a computational expensive task: each fragment requires an additional framebuffer memory read and other operations to be done:
  - Did you ever notice the fps drop when smoke, volumetric fog, or explosions are rendered?



Alpha test

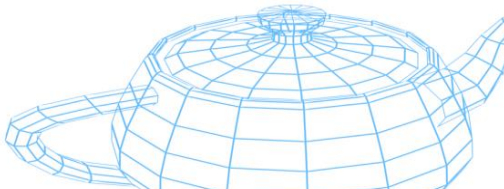
- **Hard transparency:** a pixel can be either solid or invisible:
  - Alpha test is activated through **`glEnable(GL_ALPHA_TEST)`**.

- A fragment is discarded when its alpha value is above/below a specific threshold:
  - Customize the reference threshold using the command **`glAlphaFunc(function, value)`**:
    - E.g.: **`glAlphaFunc(GL_GREATER, 0.5f);`**

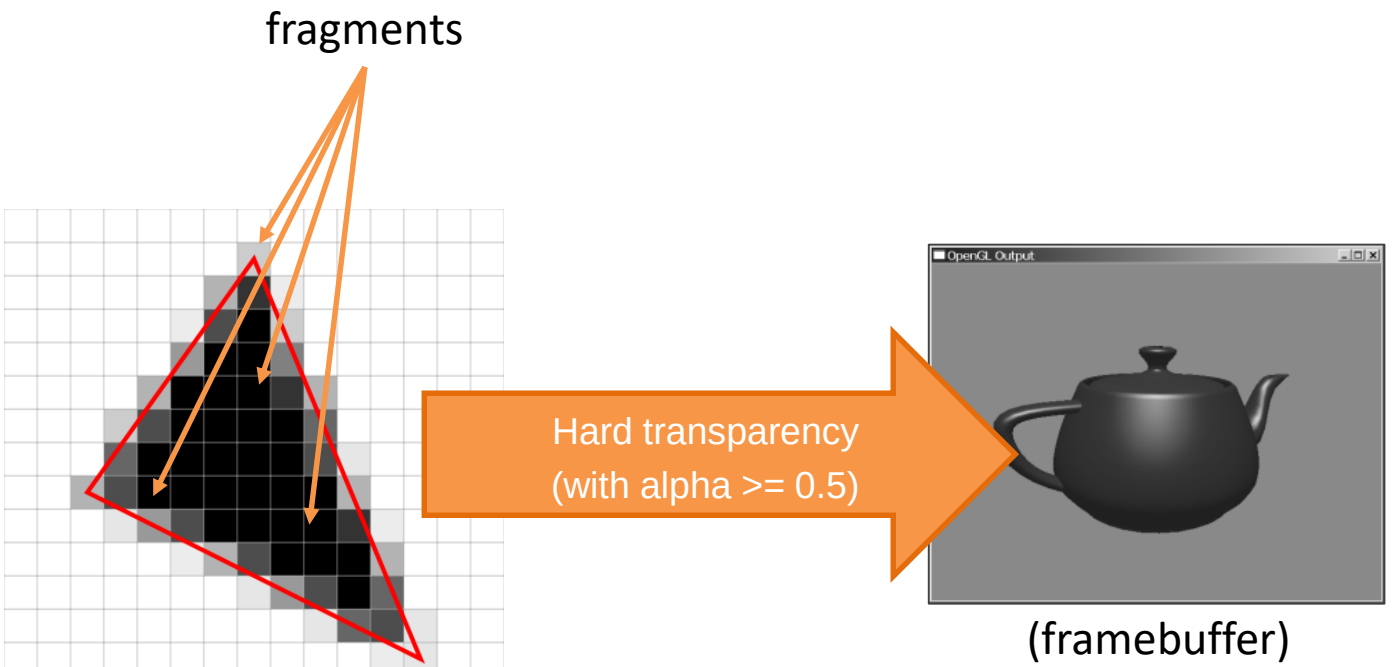
- Sorting is not required when hard transparency is used:
  - The depth test works as usual.

- In modern OpenGL, alpha testing can be directly implemented in fragment shaders, e.g.:

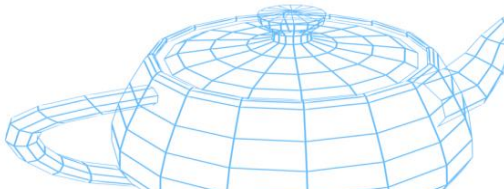
if (pixel.a < 0.5f) discard;



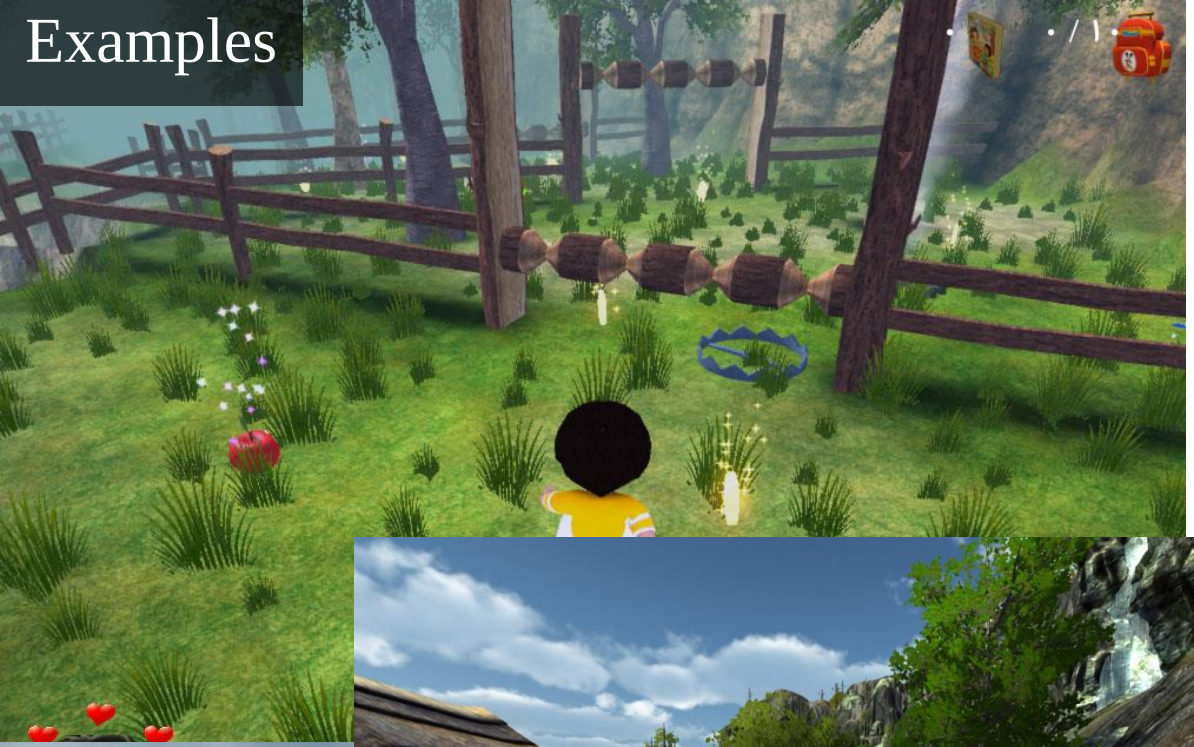
Alpha test

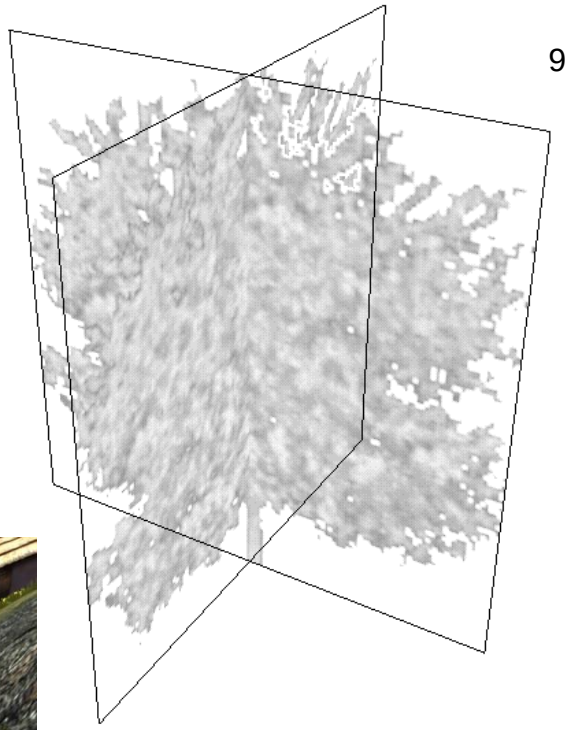


New fragments that pass the z-test and with an alpha value of 0.5 or higher replace the framebuffer content



Examples

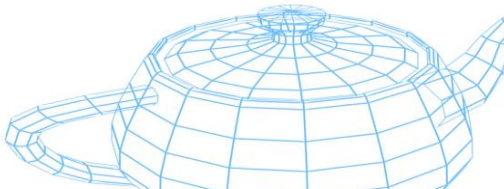





Mountain Village
NEW DEMO
Inside NeoAxis 1.1
neoaxis
engine

Blending

- Blending is used to combine values already written in the framebuffer with the fragments added by a new primitive.
- Blending is activated through the command **`glEnable(GL_BLEND)`**.
- Different options are available to specify how pixels in the framebuffer interact with the new fragments:
  - Use **`glBlendFunc(sourceFactor, destFactor)`** to change OpenGL behavior.



Blending

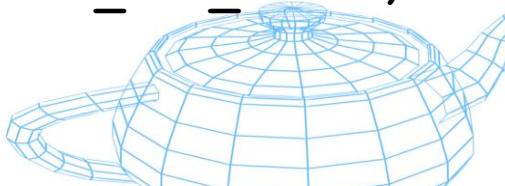
- Blending simply works as follows:

$$fbOutput = sourceFactor \times sourceColor + destFactor \times destColor$$

*(where **fbOutput** is the color of the pixel written to the framebuffer, **sourceColor** is the fragment output pixel and **destColor** is the color of the pixel already in the framebuffer)*

- Factors can be any combination of **`GL_ZERO`**, **`GL_ONE`**, **`GL_DST_COLOR`**, **`GL_ONE_MINUS_DST_COLOR`**, **`GL_SRC_ALPHA`**, **`GL_ONE_MINUS_SRC_ALPHA`**, **`GL_DST_ALPHA`**, **`GL_ONE_MINUS_DST_ALPHA`**, and **`GL_SRC_ALPHA_SATURATE`**.

- Typical transparency is implemented through the linear interpolation of two values using **`glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`**.

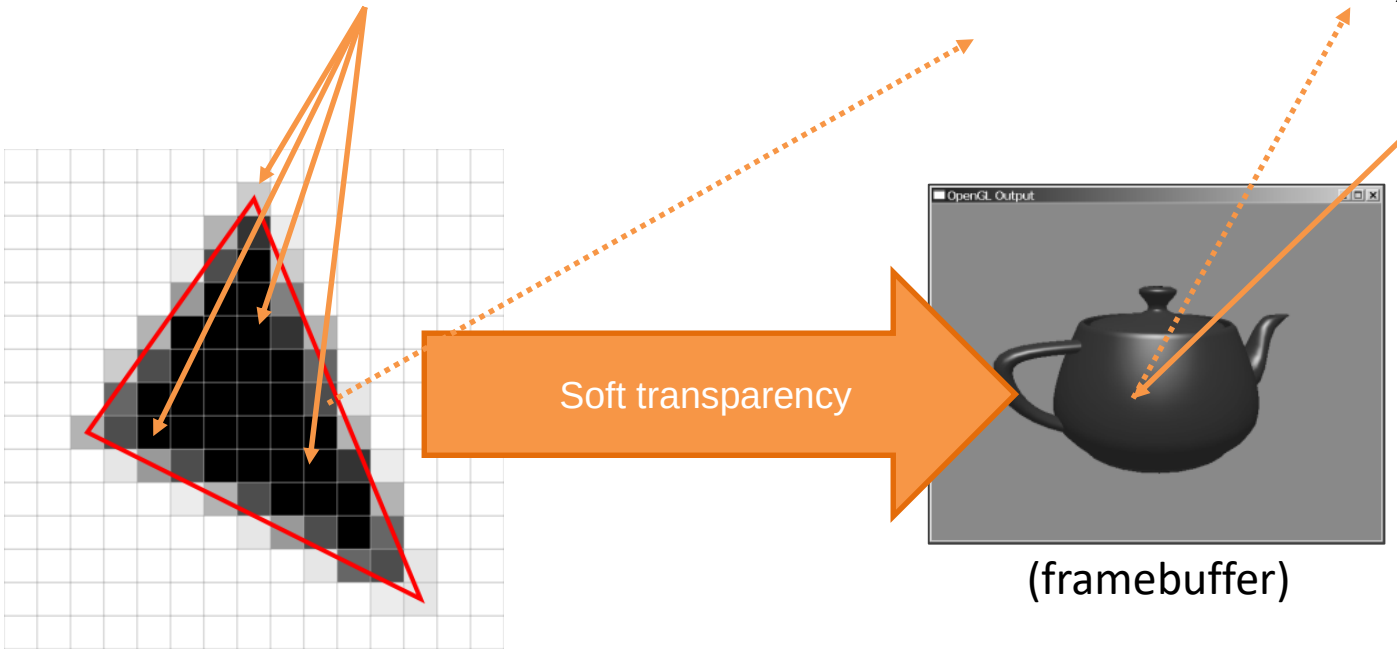


Blending

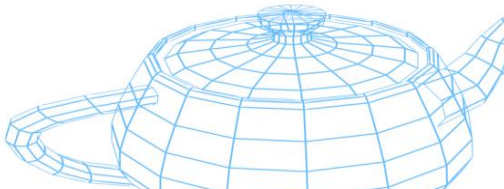
`glBlendFunc(GL_*, GL_*)`

fragments

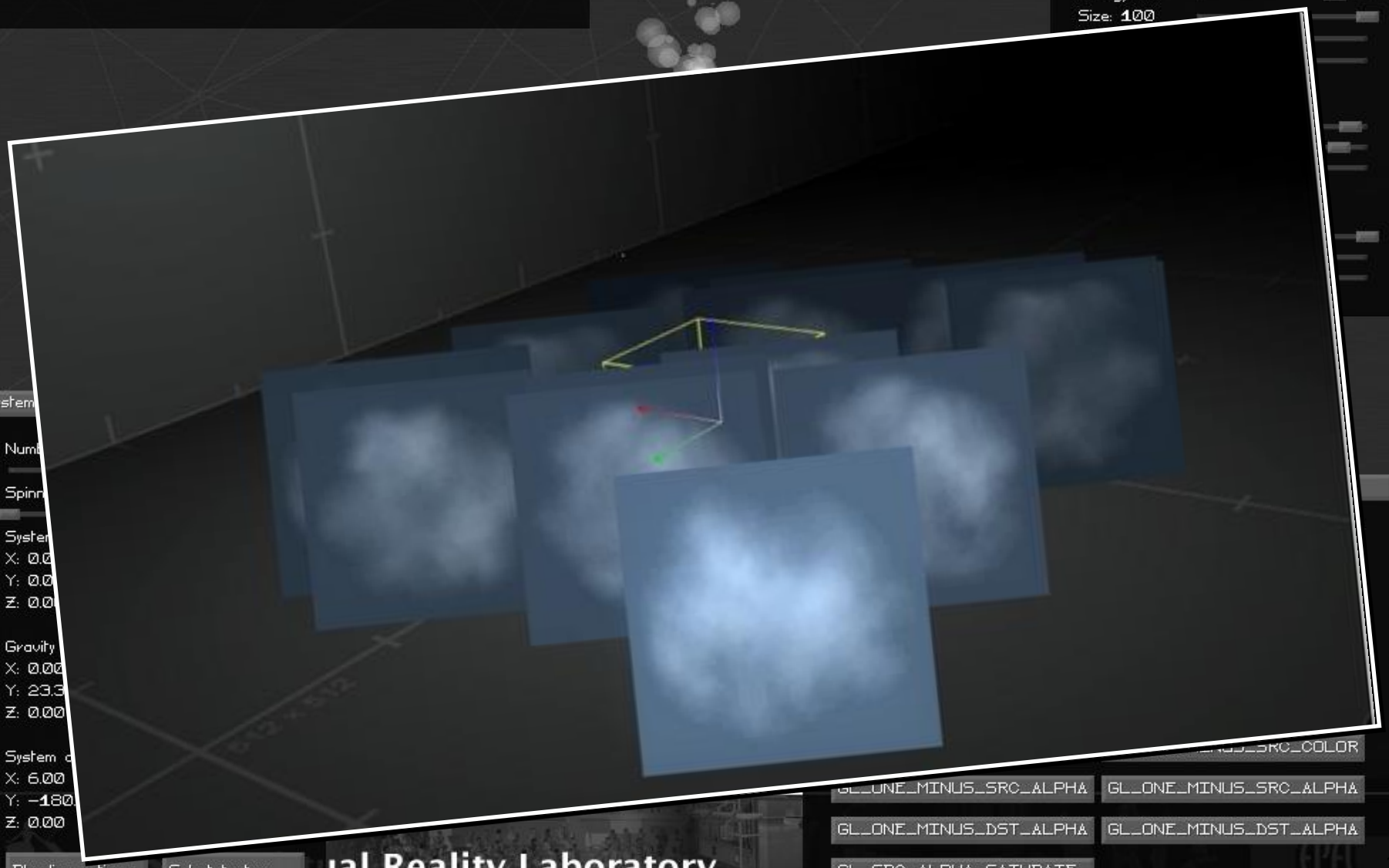
sourceFactor × fragment + destFactor × $pixel_{xy}$ = new $pixel_{xy}$



New fragments that pass the z-test replace the framebuffer content with a new pixel computed as the output of the blending function that takes the fragment color and the previous framebuffer pixel as input.



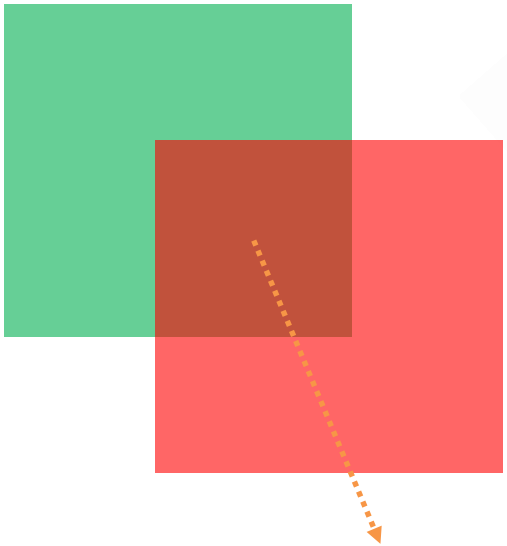
Particle engine demo



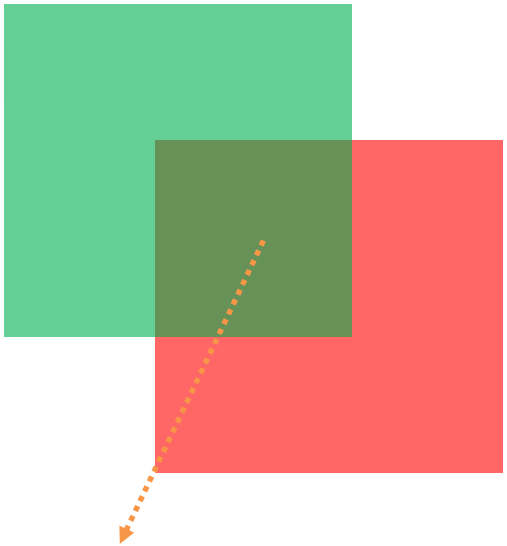
Blending problems

- Most soft transparency blending operations are order-dependent:

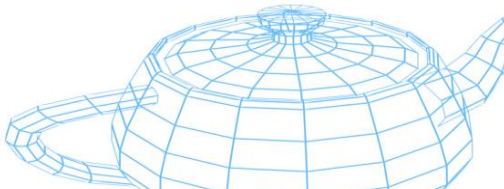
a) Green first, then red:



b) Red first, then green:



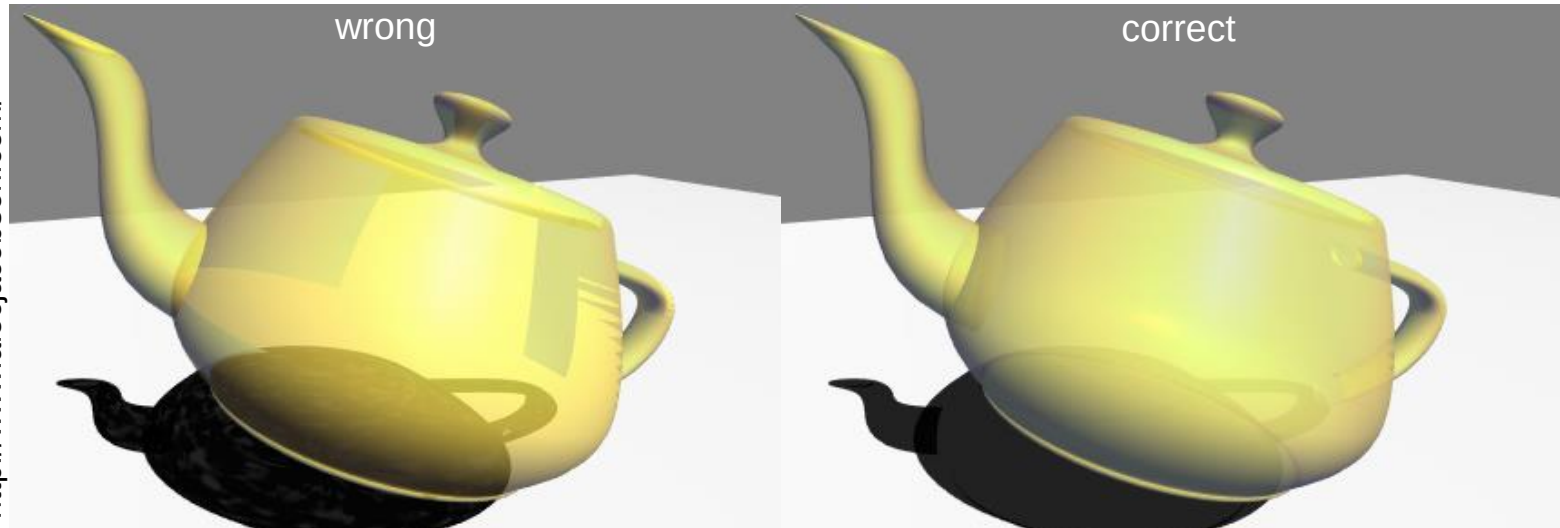
RGB: [194 83 60] != RGB: [102 147 89]



Blending problems

- Correct transparency works only when transparent triangles are rendered back to front:
  - Real-time sorting is computationally expensive on a per-triangle basis.
  - Interleaved triangles are not rendered properly.

- Mesh triangles are always passed in the same order (and are most likely rendered in the same sequence):
  - Depending on the mesh and camera orientation, their sorting can be correct or wrong.



Multi-pass rendering

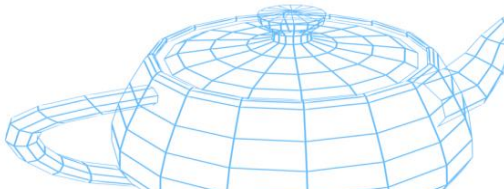
- **Single-pass rendering:** all the elements of the scene are rendered within one single loop/iteration:
  - Each object is rendered in its final state, including textures, multiple light contributions, etc.

- **Multi-pass rendering:** the scene is generated after multiple iterations, where at each step additional information is provided:
  - E.g.: rendering the full scene once per each light source available, then use additive blending to merge the output images together to generate the final image.

1 2 3 1 + 2 + 3

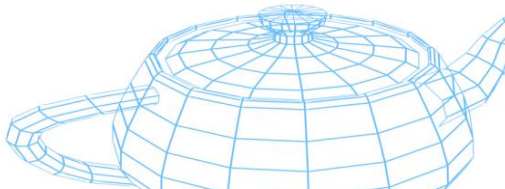
Transparency rendering

- Relatively artifact-free transparency rendering can be achieved through a multi-pass approach:
  - 1st pass:
    - Clear buffers, enable depth testing.
    - Render all the solid objects (as we did so far).
  - 2nd pass:
    - Set depth testing to read-only via **`glDepthMask()`**.
    - Depth-sort transparent objects (farthest to nearest).
    - Render each transparent object in two sub-passes:
      - Sub-pass 1: enable front-face culling.
      - Sub-pass 2: enable back-face culling.



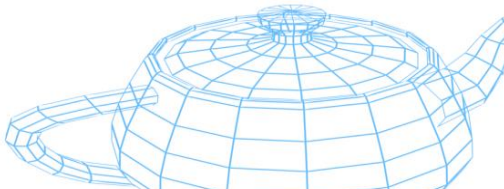
Transparency rendering

- The previous technique can be easily implemented in your graphics engine's list class:
  - The list already contains all the objects to be rendered, including their world matrix.
  - Sort transparent objects according to distance between the positional part of their world matrix and the positional part of the camera matrix:
    - Make sure to compare positions defined in the same space, e.g., world coordinates or eye coordinates.
  - You can refine results and limit cross-penetrations by applying a bounding sphere to each mesh and by including its radius in the depth sorting:
    - The OVO format includes information about each mesh's bounding radius.



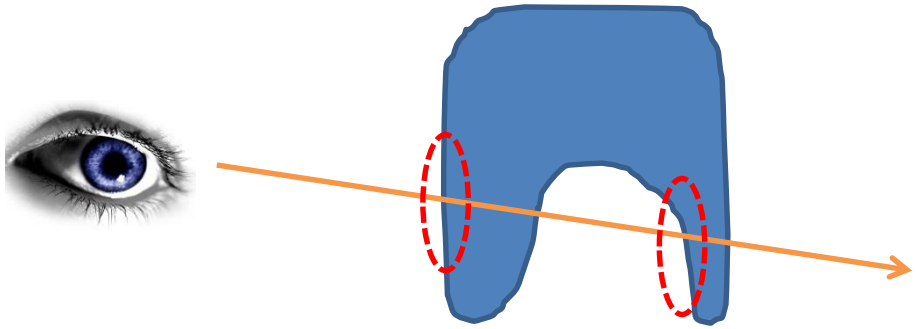
Graphics engine remarks

- Class material:
  - Add an **`isTransparent()`** method to quickly identify transparent materials.
  - Consider adding a **`setTransparency()`** method to change the alpha value of ambient, diffuse, and specular with one single instruction.
- Class list:
  - Implement a depth-sorting (back to front) method for transparent meshes.
- Class mesh:
  - If a mesh uses a transparent material, render it in two passes using different face culling parameters during each pass.
- Improve font readability by writing on top of semitransparent rectangles, e.g.:

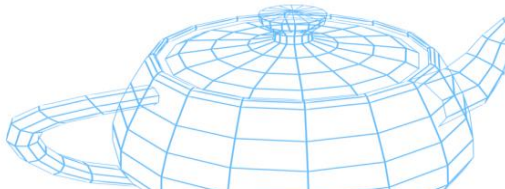


Transparency rendering

- The previous technique is a compromise between implementation complexity, rendering quality and computational power required:
  - It requires depth-sorting:
    - Although not on a per-triangle basis.
  - It does not work on nested or interpenetrating meshes.
  - It does not work on meshes with complex shapes, e.g.:

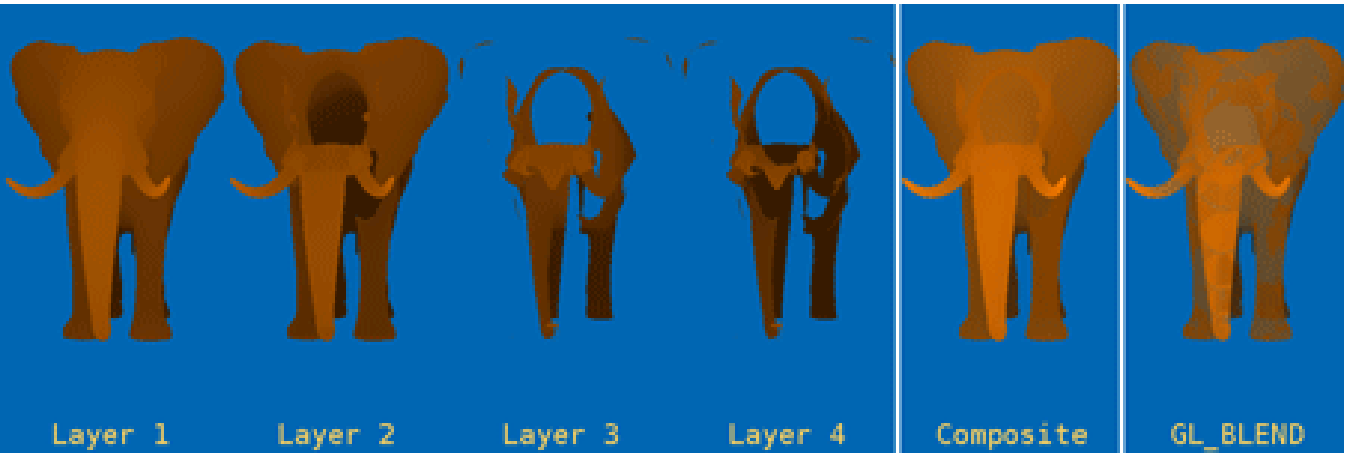


- Several Order-Independent Transparency (OIT) techniques exist (such as A-buffers and depth peeling) but require a recent version of OpenGL and additional knowledge.



Depth peeling (OIT technique)

- The idea is to use a more complex framebuffer with multiple layers (each one with its own depth buffer).
- Multi-pass rendering is used (one pass per layer):
  - The scene is rendered.
  - Only the nearest pixels are stored in the first layer (as we did so far).
  - Rendering is then done again, once for each layer:
    - At each iteration, the 2nd, 3rd, …, nth nearest pixels are stored within each layer.
- Once all the layers have been processed, they are rendered back to front.



http://www.alecjacobson.com/