Game Architecture and best practices

Reinhold Preiner

Thomas Weber

Institute of Computer Graphics and Algorithms

Vienna University of Technology

Topics for today

- Software Architecture for Game Development
  - Rules of software engineering
  - Entity Management
  - Scene-Representation
  - Basic Game-Lifecycle
  - The Render-Loop
  - Resource Management
- Best Practices and Useful Tips

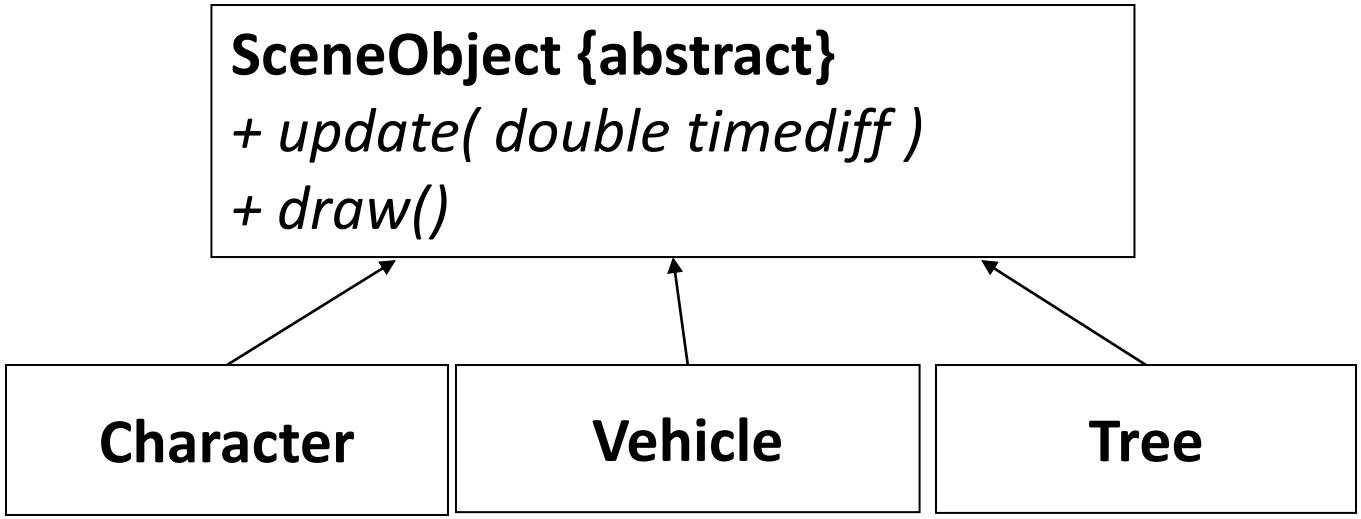
Software Architecture for Game Development

Some Rules of Good SW-Engineering

- **SCALABILITY**
  - Be able to extend your System with minimal effort
- **MAINTAINABILITY**
  - Be able to change/adapt your System with minimal effort
- **MODULARITY**
  - If you have to do the procedure in different corners of your System, do it once in one module (class) and call it wherever you need it.

- These Concepts interact with each other
- But: Beware of Overengineering -> you only have one Semester!

Entity management

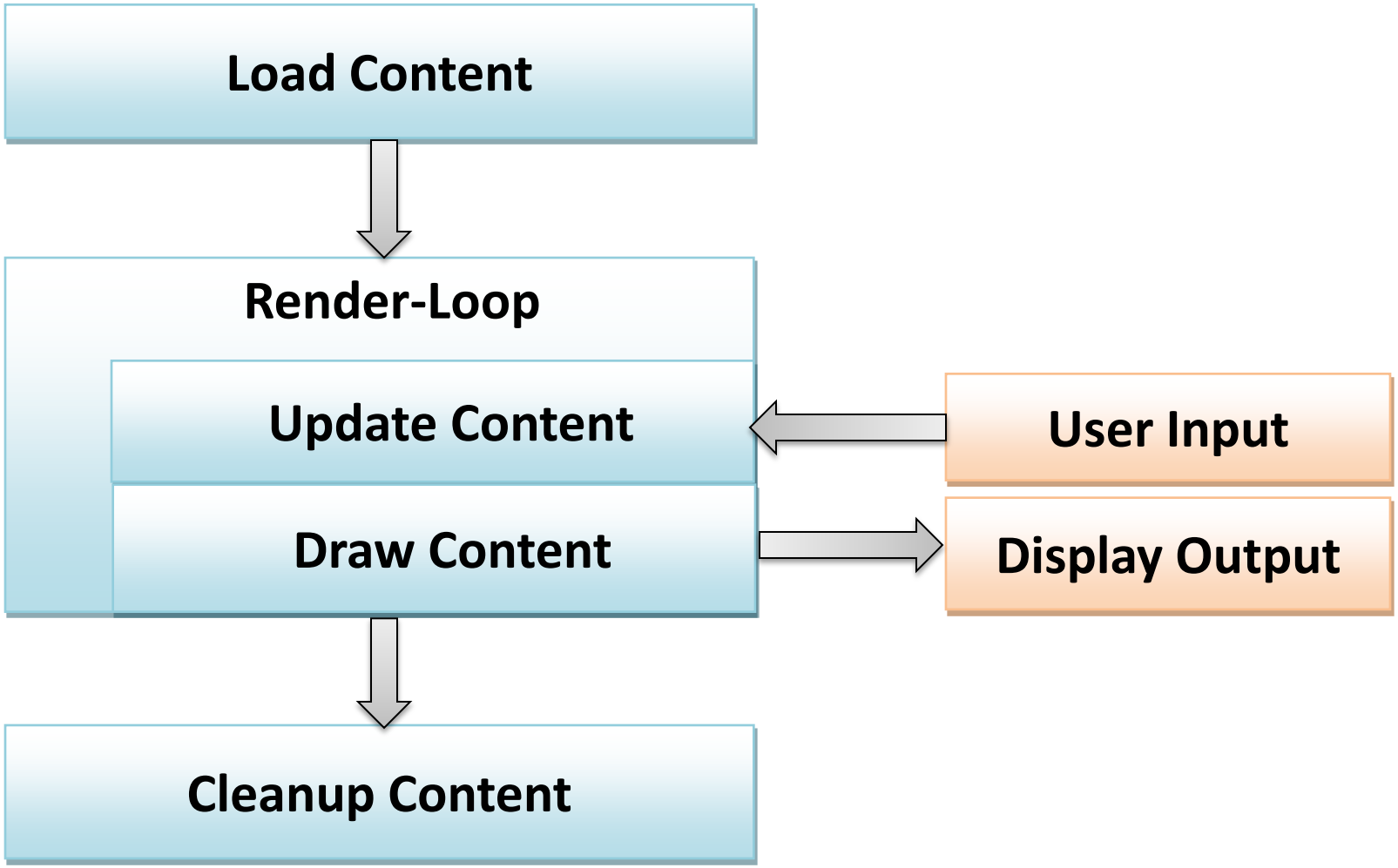


- Character::update() handles user input
- Vehicle::update() controlled by AI
- Tree::update() does nothing

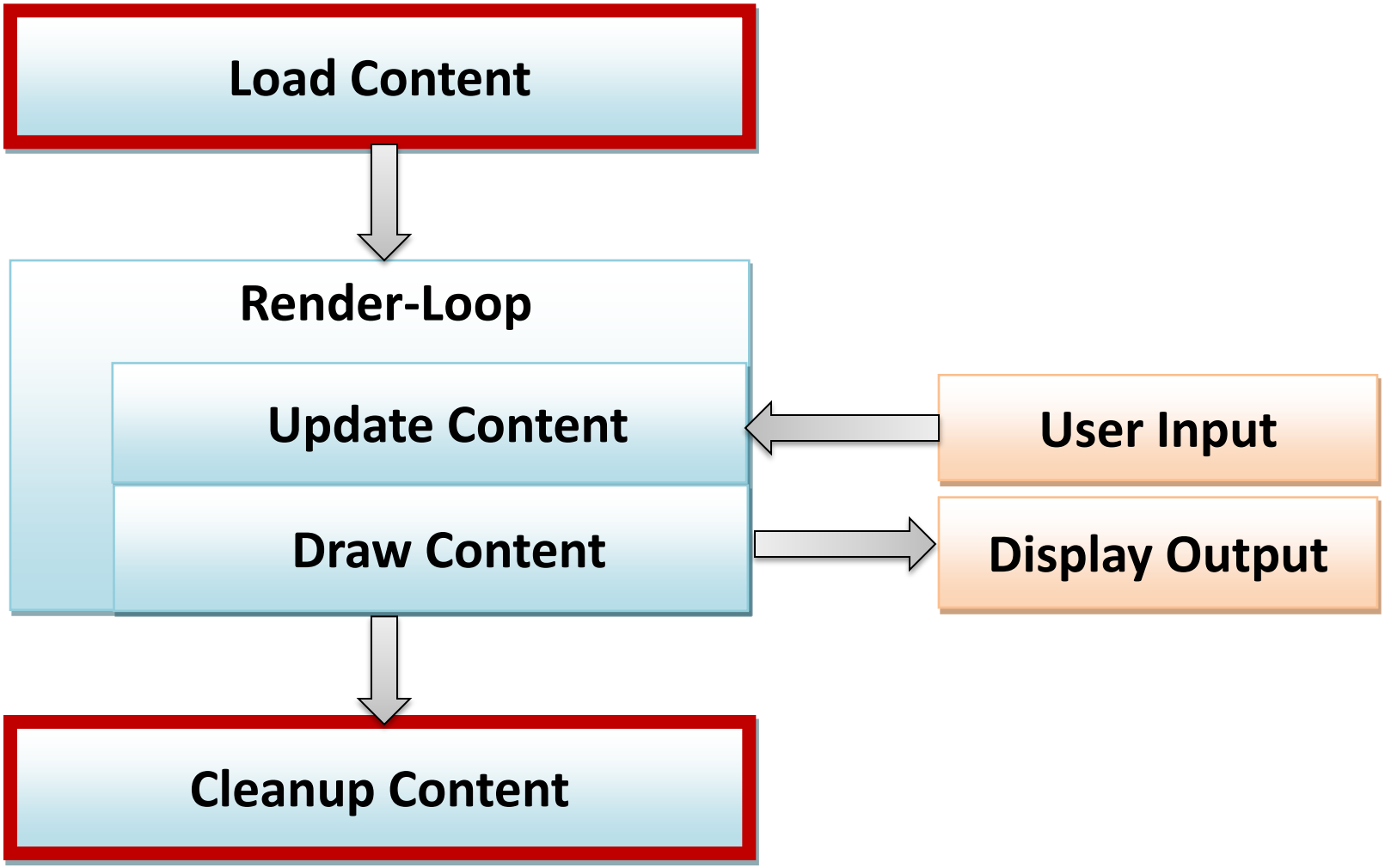
Scene-Representation

- Which data-structure for object organization?
- Depends on your needs
- 2 Basic approaches:
  - Linear List (e.g. std::list<SceneObject*>)
    - Advantage: Easy to implement
    - Disadvantage: Bad for hierarchical scenes
  - Scene-Graphs:
    - Ideal for hierarchical animations (e.g. solar system)
- You can also use both (or none ;))

Basic Game Lifecycle



Resource-Management



Resource-Management

- Common Design-Concepts:
  - RAII
  - Resource managing module

RAII

- “Resource Acquisition Is Initialization”
- Constructor: Resource acquisition
  - Textures, shader, meshes etc.
  - Destructor: Release resources
- Advantages:
  - Simplifies programming work
  - Exception safe
- Disadvantage:
  - Beware of shallow copies

    (see C++ talk / part 2)

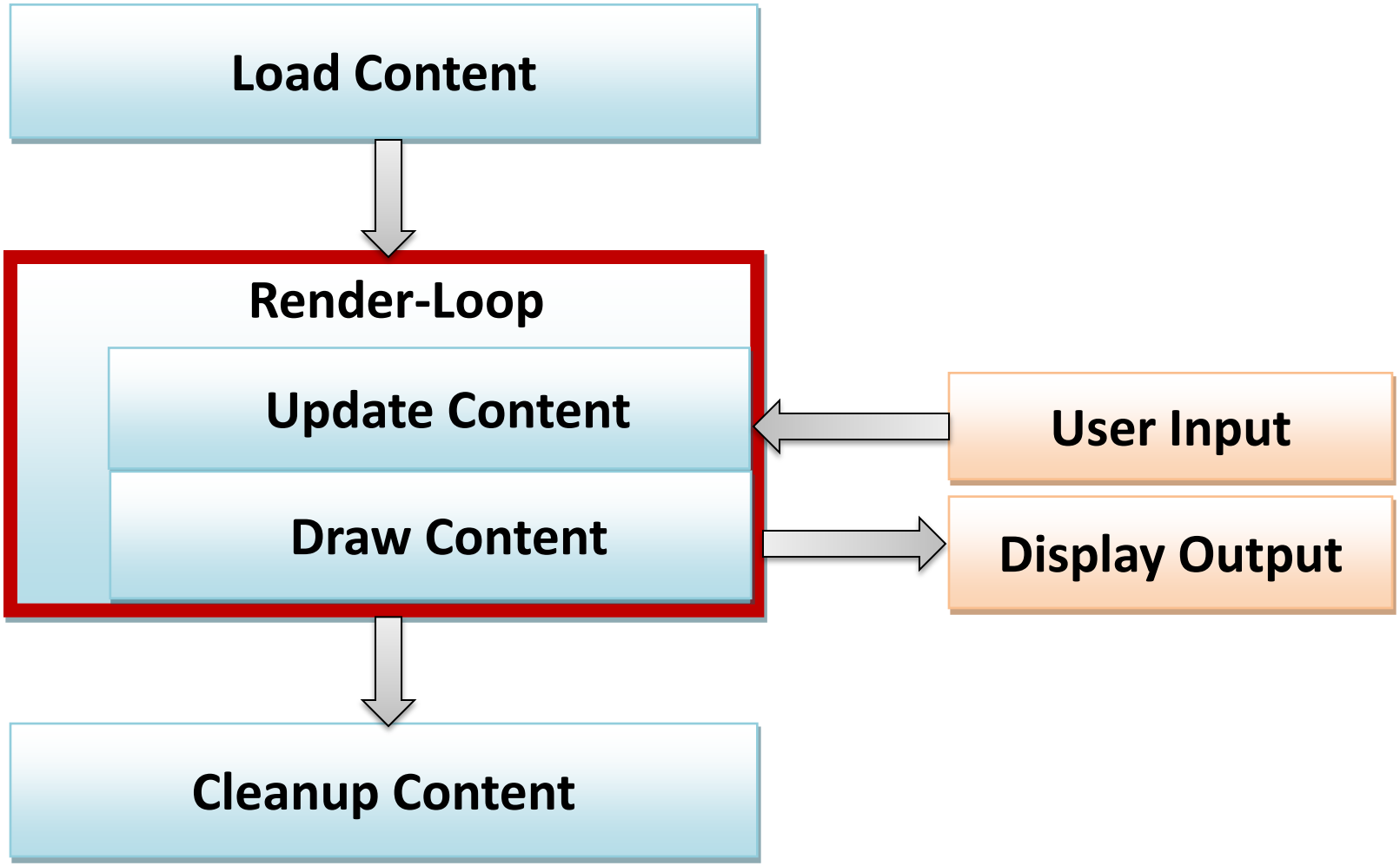
Resource-Managing-Module

- Central resource manager class handles all resources
- Has methods like: loadMesh, loadTexture, etc.
- Loads resources once and returns references
- Internally stores resources in maps
- Could also support reference counting
- Handles cleanup at application end

Resource-Managing-Module

```
Resource* loadResource( "resourceName" )
{
    if ( "resourceName" already loaded  )
        return lookup( "resourceName" );
    else
    {
        res = loadFromFile( "resourceName" );
        storeInMap( "resourceName", res );
        return res;
    }
}
```


Render-Loop


Load Content
Render-Loop
Update Content
Draw Content
User Input
Display Output
Cleanup Content

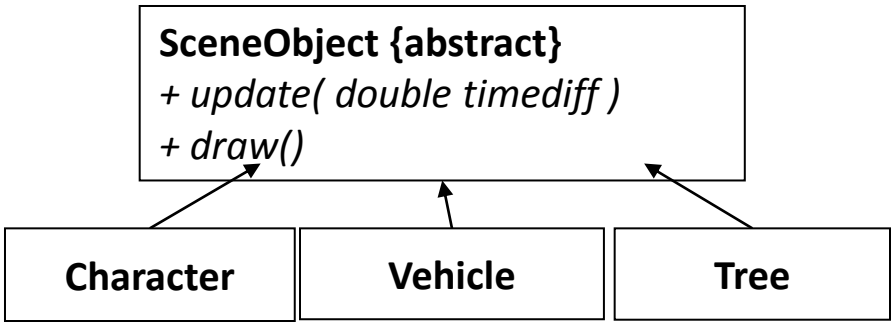
Render-Loop

```
while( <running> )
{
    double dt = <Calculate time Delta to prev. Frame>;

    foreach SceneObject* so in sceneObjectList
        so->update( dt );

    foreach SceneObject* so in sceneObjectList
        so->draw ();

    <Swap Buffers>
}
```



Best Practices and Useful Tips

OpenGL C++ Wrappers

- Wrapper objects for OpenGL functions
- For things like textures or shaders
- Example usage

```
Shader shader(“my_shader”);
Mesh mesh(“teapot.mesh”);
shader.bind();
shader.setUniform(“color”, vec3(1,0,0));
mesh.draw(&shader);
shader.unbind();
```

- Write code before implementing wrappers

Config-Loading

- Don’t hardcode constant values
- Use a config file
- Example syntax:

```
lightColor 1 0.5 0.5
lightPosition 10 10 30
shaderName “shader/myShader”
```

- Tweak values without recompilation
- Example Config class:

```
Config config(“game.config”);
String shaderName = config.shaderName;
```


Time Measurement

- Most frameworks support timer functions
- Clamp longer time differences
  - Useful for debugging, pause state, etc.

```
double last = 0.0, logicTime = 0.0;
while (running) {
    double now = glfwGetTime();
    double dT = min(config.maxDT, now-last);
    last = now;
    logicTime = logicTime + dT;
}
```


FPS computation

- Each frame do

```
++frames;
if (now – last > config.framerateUpdateTime) {
     msPerFrame = 1000.0 * (now-last) / frames;
     last = now;
     frames = 0;
}
```

- Encapsulate in function or class
- FPS = 1000 / msPerFrame
- ms/frame is preferable metric
- FPS just better known

Input handling

- Polling for time spanning actions
  - Pressing forward key
  - Firing machine gun

```
glfwGetKey('W');
```

- Events for single key hits
  - ESC, F-keys, etc.
  - Firing grenade
  - Execute action

```
void GLFWCALL keyCallback(int key, int state) {…}
glfwSetKeyCallback(keyCallback);
```