Seminar IV

How to start a semestral project

Creating a scene

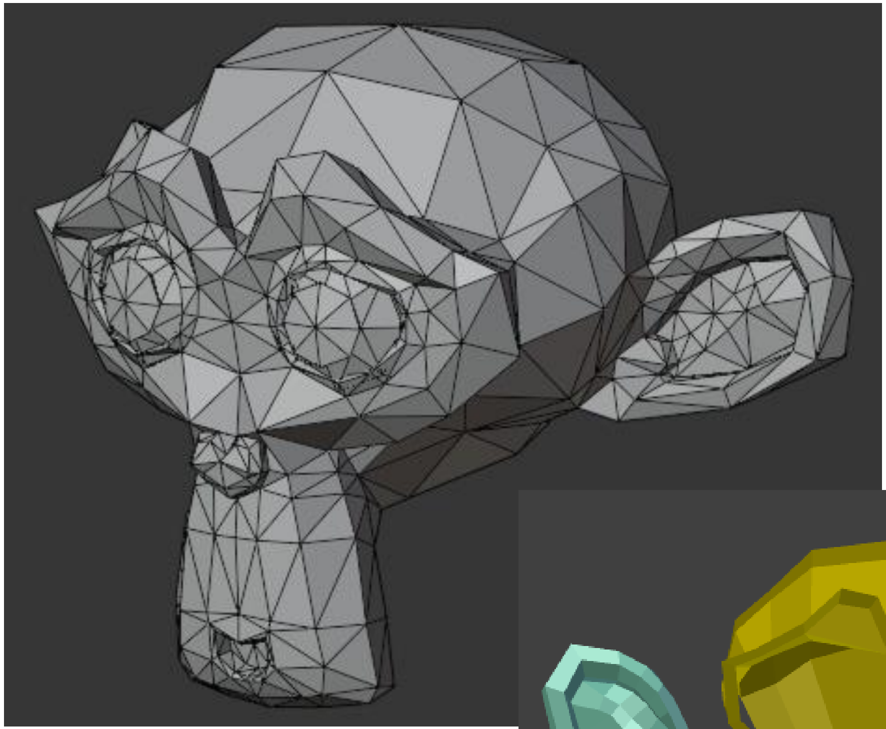
- unit size models located at the origin

  $\Rightarrow$ easy placement in the scene
- static part
  - one big model $\Rightarrow$ exported in one file
  - models repeated in the scene $\Rightarrow$ each model a separate file $\Rightarrow$ creating multiple instances
- dynamic objects $\Rightarrow$ separate models
- procedurally generated models
- models instantiation
  - common geometry (vao, vbo, ebo)
  - instance dependent data (position, size, velocity…)
- common shaders for the whole scene x specialized shaders
- models creation – loading from files via assimp x procedural

Shaders

```c
typedef struct _ShaderProgram {
  // identifier for the shader program
  GLuint program;     // = 0;

  struct {
    // vertex attributes locations
    GLint pos;        // = -1;
    // uniforms locations
    GLint PVMmatrix;  // = -1;
    // ...
  } locations;

  // ...
} ShaderProgram;
```

Scene objects geometry



single mesh

x

multiple meshes

taken from docs.blender.org

Storing the geometry of scene objects

```
typedef struct _ObjectGeometry {
  GLuint vertexBufferObject;  // vbo identifier
  GLuint elementBufferObject; // ebo identifier
  GLuint vertexArrayObject;   // vao identifier
  unsigned int numTriangles;  // number of triangles in the mesh

  // material properties
  glm::vec3     ambient;
  glm::vec3     diffuse;
  glm::vec3     specular;
  float         shininess;
  GLuint        texture;

 // geometry/object specific data
} ObjectGeometry;
```

one mesh
x
multiple meshes

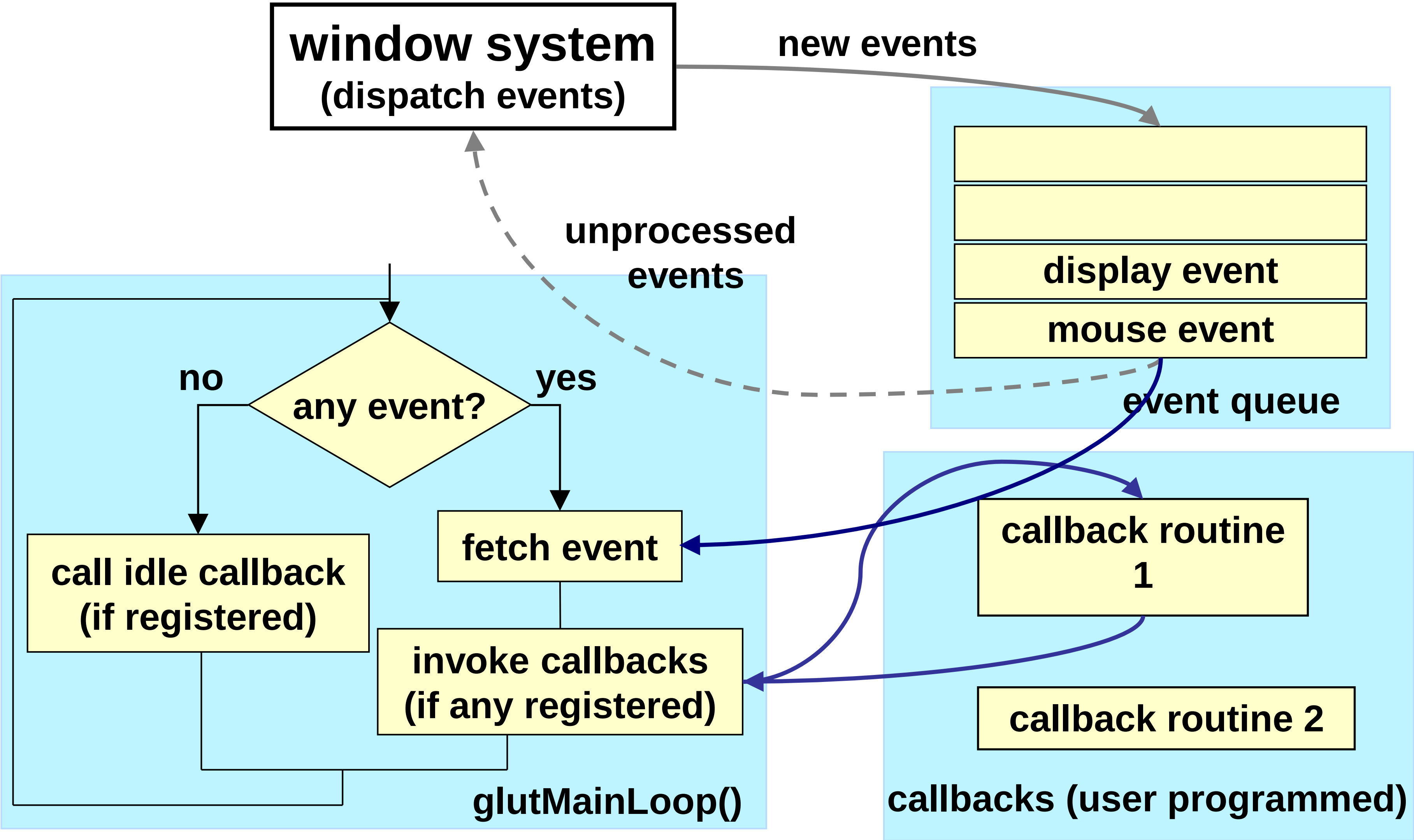
Object instantiation

```
struct ObjectInstance {
  ObjectGeometry* geometry;
  glm::mat4       modelMatrix;

  // dynamic object data => used to derive model matrix
  // glm::vec3 position;
  // glm::vec3 direction;
  // float     speed;
  // float     size;

  // ...

} ObjectInstance;
```

Application skeleton using GLUT

- see source code in skeleton.zip

Event processing through callbacks



Processing key combinations

```
// keys used in the key map
enum {
 KEY_LEFT_ARROW, KEY_RIGHT_ARROW,
 KEY_UP_ARROW, KEY_DOWN_ARROW,
 KEY_SPACE,
 KEYS_COUNT
};
// key map
bool keyMap[KEYS_COUNT];

// handling of events based on the key map
if(keyMap[KEY_RIGHT_ARROW] && keyMap[KEY_UP_ARROW])
  moveRightUp();
```

Processing key combinations

```
// set up callbacks
glutKeyboardFunc(keyboardCallback);
glutKeyboardUpFunc(keyboardUpCallback);
glutSpecialFunc(specialCallback);
glutSpecialUpFunc(specialUpCallback);


// special key released callback
void specialUpCallback(int releasedKey, int mouseX, int mouseY) {
  switch (releasedKey) {
    case GLUT_KEY_RIGHT:
      keyMap[KEY_RIGHT_ARROW] = false; break;
    //…
    }
  }
}
```

Simple animation using timer callback

```
void timerFunc(int id) {
    // possible processing of id value
    glutTimerFunc(33, timerFunc, 0);        /* register new animation step */

    spin += spinStep;                       /* set new angle for rotation */

    glutPostRedisplay();                    /* redraw window */
}


int main(int argc, char **argv) {
    …
    glutTimerFunc(33, timerFunc, 0);        /* register this event after 33ms */
    …
    glutMainLoop();                         /* finally, enter event loop */
}
```

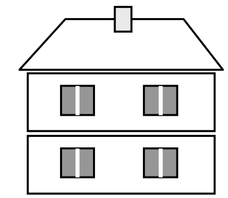
Object representation in the code

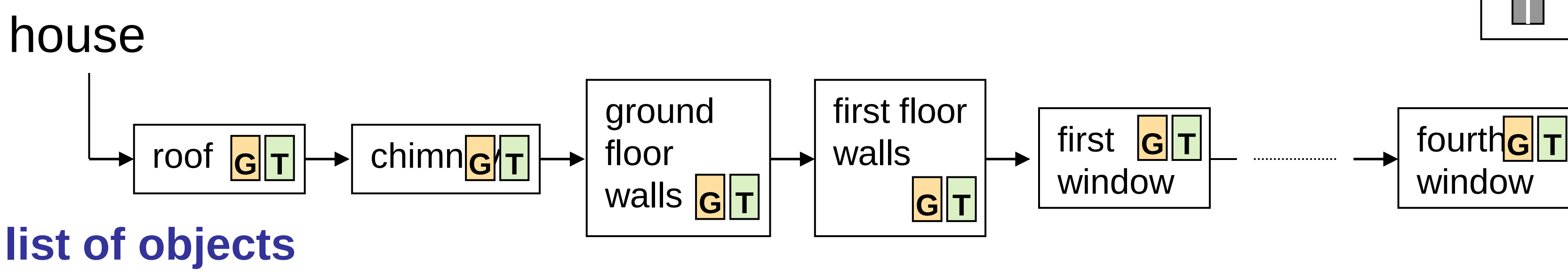
- object-oriented approach x traditional procedural approach

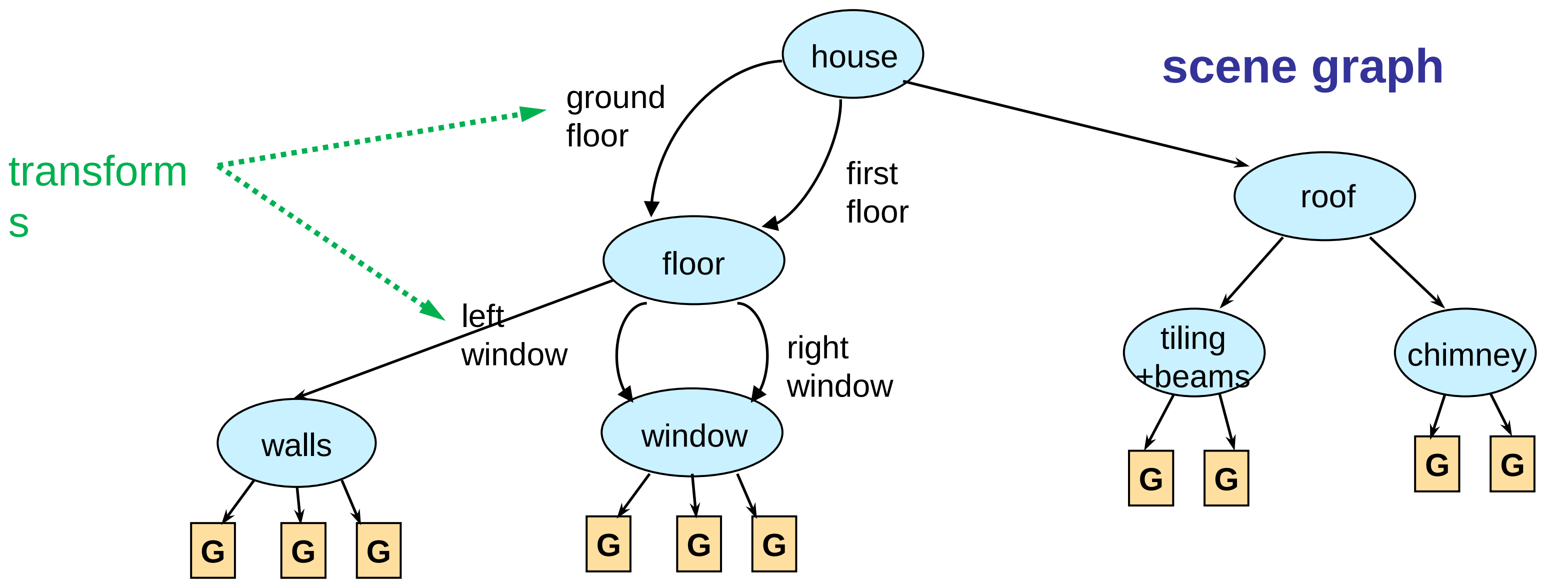
```
class ObjectInstance {

protected:

  ObjectGeometry* geometry;

  glm::mat4 localModelMatrix;

public:

  virtual void update(float elapsedTime) {
    // update model transformation - localModelMatrix
    // ...
  }

  virtual void draw(const glm::mat4& viewMatrix, const glm::mat4& 
projectionMatrix) {
   // draw node geometry using globalModelMatrix
   // ...
  }
```

Scene structure representation

- linear (list of objects) x hierarchical (scene graph)

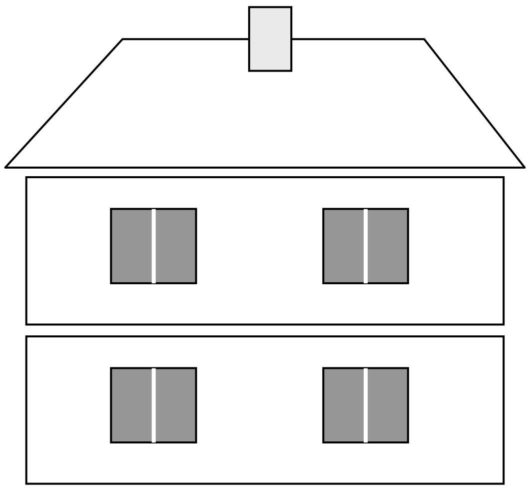


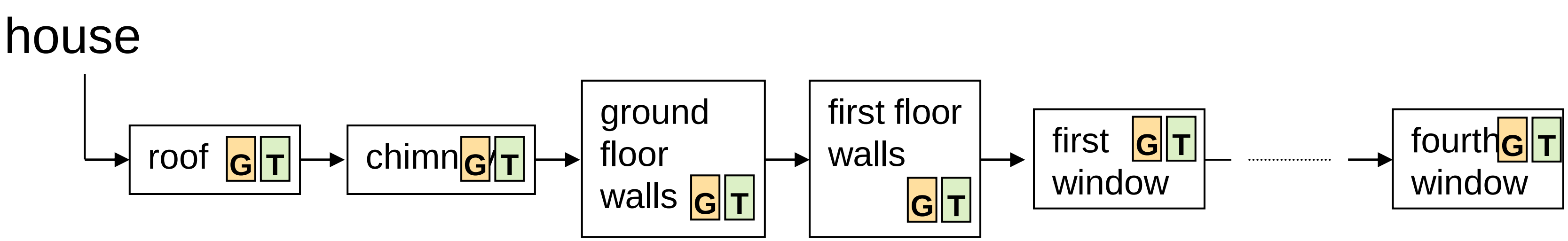




Nonhierarchical approach

- geometric object is represented as a sequence of segments (linear data structure)
- each segment contains
  - definition of graphical elements and their attributes
  - transformations





- this modeling technique contains no information about relationship among objects (*i.e. model/scene logical structure is not known/expressed*)

  ⇒ we can easily manipulate individual segments of the model but more complex structures like ground floor are hard to reach

Nonhierarchical approach (contd.)

- how to encode this data structure in your program?

  ⇒ each segment is represented by a drawing function (including transformations)

```
void house(void) {          /* the whole model of the house */
  roof();                   /* roof segment */
  chimney();
  groundFloorWalls();
  firstFloorWalls();
  firstWindow();
  …
  fourthWindow();
}
```

```
void roof(void) {
  T /* set transformations */
  G /* set attributes like color, etc */
    /* draw geometric primitives */
}
```

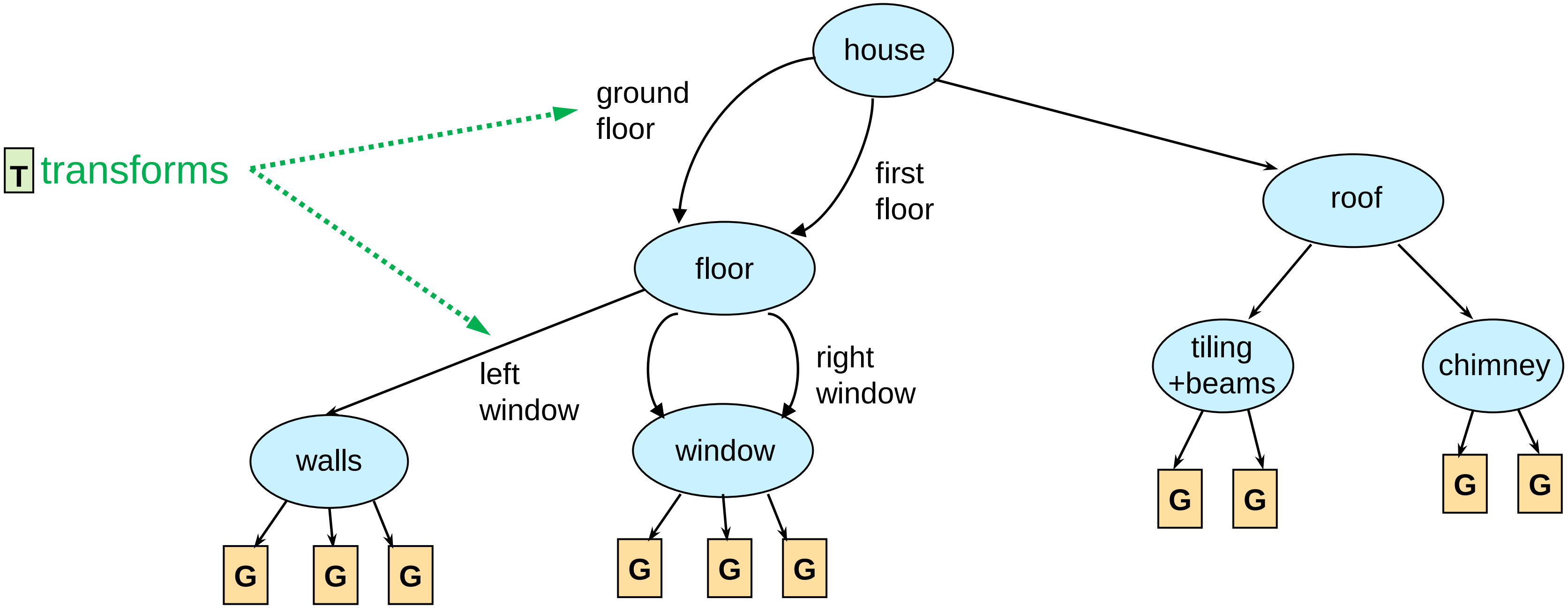
Hierarchical models

- we can represent the relationship among segments of the model with graphs
- most common type of graph is **connected tree** (*directed graph without closed path or loops and every node, except the root, has a parent node*)

  ⇒ represents very well the logical model structure (i.e. composition of the model)
- typically:
  - leaves represent geometry (graphical primitives)
  - internal nodes represent superior/parent segments (details)
    (*e.g. roof, ground floor, first floor, etc.)*
- **drawing** an object described **by** a tree requires performing **a tree traversal**
  - depth-first search
  - breadth-first search
- the **same algorithm for traversing** graph in a program **should be always used** ⇒ the same result/model
- attributes are inherited or redefined by the nodes

Directed acyclic graph (DAG)

- internal nodes represent superior/parent segments (details)
- leaves represent graphical primitives
- relationship between nodes is represented by edges (parent-children relation)
- transformations are used to label the edges of the graph
- transformations represent the incremental change when we go from the parent to the child



Directed acyclic graph (contd.)

how to encode this tree structure into a code sequence?

- internal nodes ⇒ store references to children
- edges ⇒ attributes and transformations of children T
- leaves ⇒ represented by drawing functions G
- drawing an object ⇒ requires performing a tree traversal

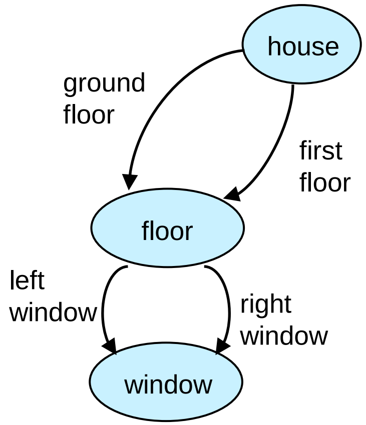
encoded into function

```
/* model of the whole house */
void house(void) {
T /* set transformations for the ground floor */
  floor();
T /* set transformations for the first floor */
  floor();
T /* set transformations for the roof */
  roof();
}
```

```
/* model of the window */
void window(void) {
  /* draw window geometry */
G drawGeometry();
}
```

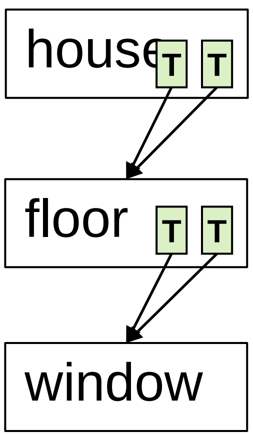
static version

Where to store transformations?

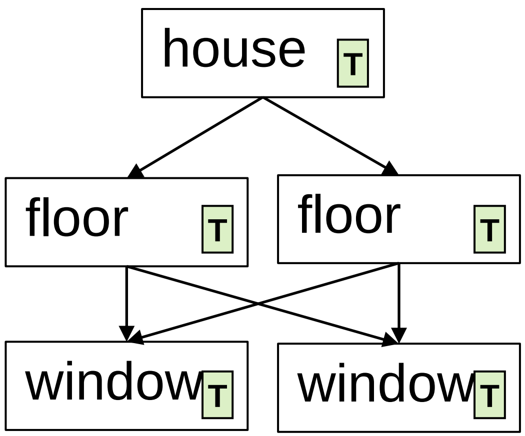


A. tranformation in parent
B. transformation in child
C. transformation on edge ⇒ represented as an extra node

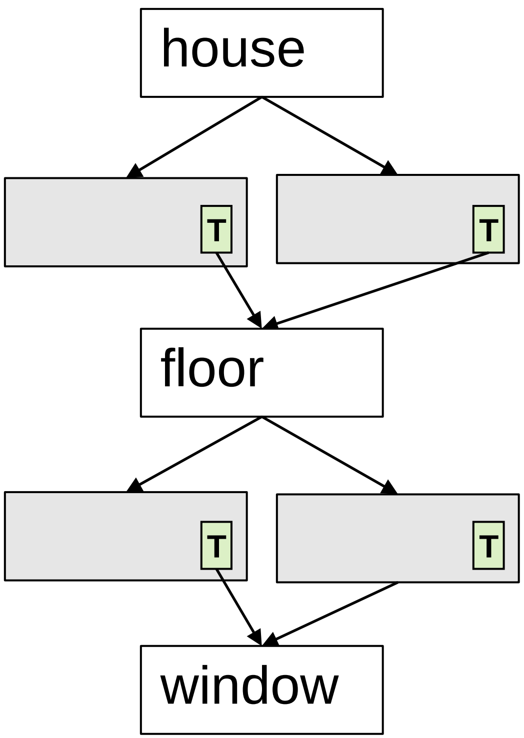
A



B



C



Directed acyclic graph (contd.)

how to encode this tree structure into a code sequence?

- internal nodes ⇒ class storing references to children + transformations T
- leave nodes ⇒ class drawing geometry G

```
class Node {
    /* common data  */
    virtual void draw() {
        /* node specific behavior  */
    }
}
```

```
class innerNode : public Node {
    /* incremental transformation */
T   glm::mat4 modelMatrix;
    void draw() {
        /* process child nodes */
    }
}
```

```
class leafNode : public Node {
    void draw() {
G   /* draw model geometry */
    }
}
```

dynamic version