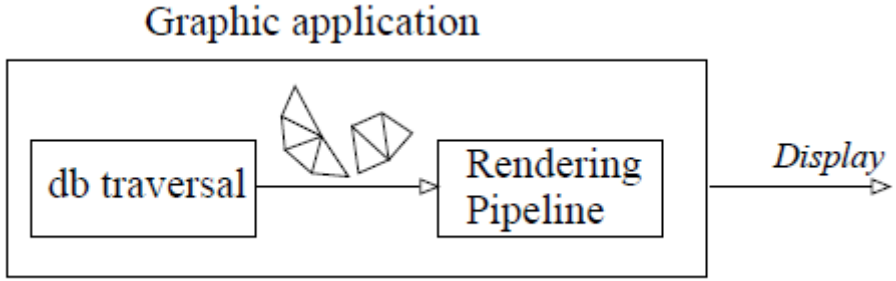
# Grafica al calcolatore
# -
# Computer Graphics

11 – Gestione delle scene

# Applicazione grafica

- Possiamo vedere l'applicazione grafica come un front-end per il motore di rendering (pipeline), costruita per alimentare (di poligoni) quest'ultima.




- L'applicazione grafica ha, tra gli altri:
  - il compito di costruire e visitare la struttura dati che rappresenta la scena, dove le correlazioni naturali tra le primitive sono rese esplicite (p.es. raggruppate in oggetti).
  - il compito di inviare alla pipeline di rendering un carico di poligoni da disegnare adeguato alle capacità di quest'ultima.

# Gestione della scena

- Nei nostri programmini abbiamo semplicemente considerato o i triangoli o oggetti senza definire relazioni, mandando tutta la scena al sistema grafico
- Ma nelle applicazioni può essere necessario fare operazioni sulla scena che rendono utile una sua strutturazione
  - Selezione di parti della scena visibili da mandare alla pipeline grafica (visibility culling)
  - Animazione di oggetti articolati
  - Interazione e detezione di collisioni
  - Simulazione fisica

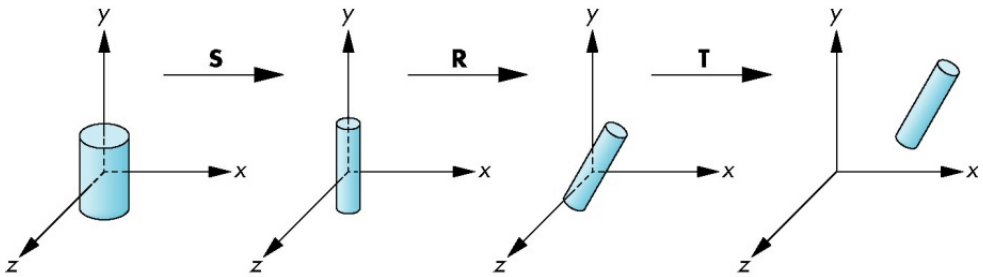
# Strutture gerarchiche

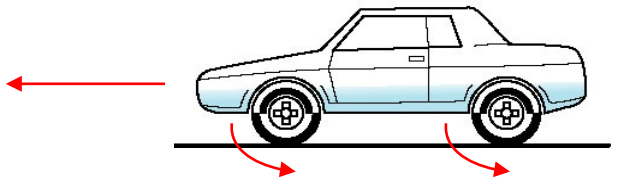
- La struttura dati che contiene le primitive della scena è bene che ne renda esplcite le naturali correlazioni
- La correlazione tra primitive può esprimersi sia tra i poligoni (raggruppamento in oggetti), sia per gli oggetti stessi (raggruppamento degli oggetti)
- Spesso tali raggruppamenti prendono la forma di gerarchie, utilizzando o alberi o grafi. Si avranno essenzialmente tre tipi di raggruppamenti
  - Gerarchie di poligoni: i poligoni della scena vengono distribuiti su una struttura gerarchica (es. ad albero).
  - Gerarchie di oggetti: esprimono relazioni di contenimento (la bottiglia nel frigo nella cucina nella casa nella città ...) oppure possono modellare oggetti articolati
  - Gerarchia della scena: (scene graph) in tal caso la scena è descritta da una gerarchia che contiene sia gli oggetti da disegnare, che alcuni stati che determinano come disegnarli (trasformazioni, shading etc..). Per fare il rendering della scena basta visitare in modo opportuno tale struttura

# Istanze e oggetti

- Possiamo avere diversi prototipi di oggetto (simboli)
- Che compaiono nella scena in diverse istanze
  - Caratterizzate da scala posizione rotazione
  - Abbiamo per ogni istanza una matrice di trasformazione
  -
- Un modello può essere composto da varie istanze di oggetti con associato un numero e I parametri
  - Symbol-instance table

| Symbol | Scale | Rotate | Translate |
|---|---|---|---|
| 1 | $s_x, s_y, s_z$ | $\theta_x, \theta_y, \theta_z$ | $d_x, d_y, d_z$ |
| 2 | | | |
| 3 | | | |
| 1 | | | |
| 1 | | | |
| . | | | |
| . | | | |



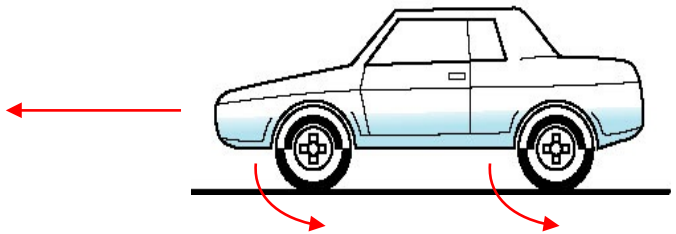



# Problema

- Non si vedono le relazioni tra le parti
  - Auto: telaio + 4 ruote
  - Due prototipi di oggetto
  - Conviene vedere come grafo

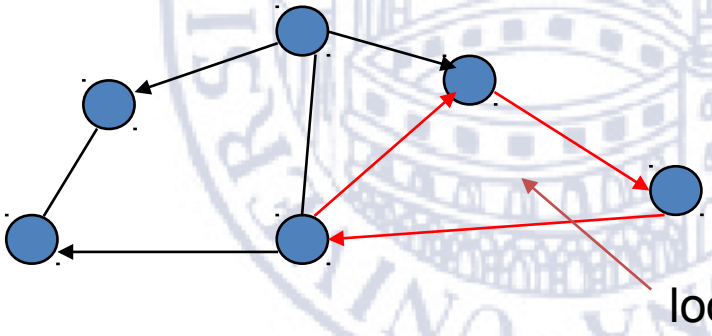
| Symbol | Scale | Rotate | Translate |
|---|---|---|---|
| 1<br>2<br>3<br>1<br>1<br>.<br>. | $s_x, s_y, s_z$ | $\theta_x, \theta_y, \theta_z$ | $d_x, d_y, d_z$ |

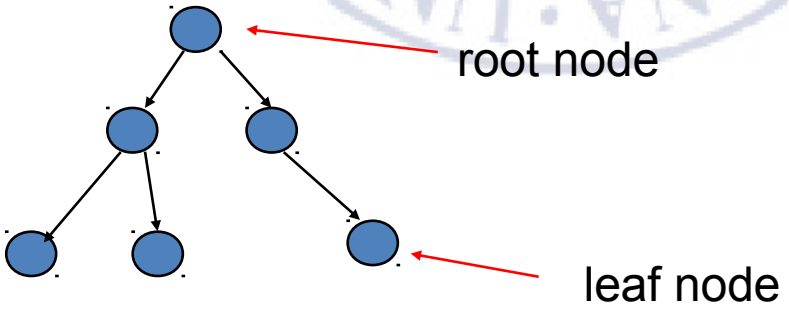
```
car(speed)
{
    chassis()
    wheel(right_front);
    wheel(left_front);
    wheel(right_rear);
    wheel(left_rear);
}
```

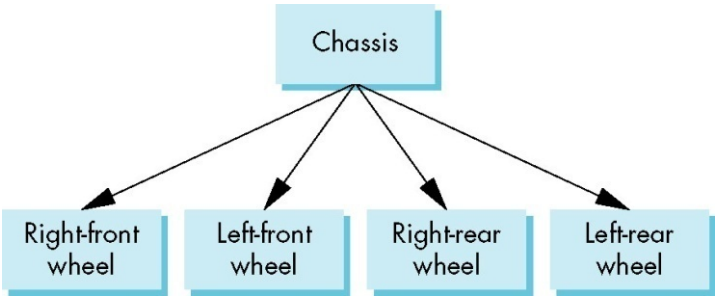


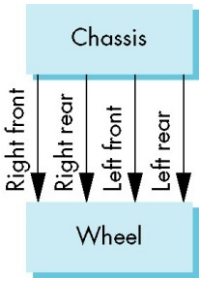
# Grafi, alberi, DAG

- Grafo:
  - Insieme di nodi e collegamenti
  - Diretto o indiretto
  - Ciclo: percorso che torna allo stesso nodo
- Albero
  - Grafo  in cui I nodi (tranne uno, la radice) hanno un genitore
  - Possono avere più figli
  - Foglia: nodo terminale
- Grafo Aciclico Diretto (DAG)
  - I nodi rappresentano gli oggetti astratti
  - Quindi se ho più istanze le lego allo stesso nodo

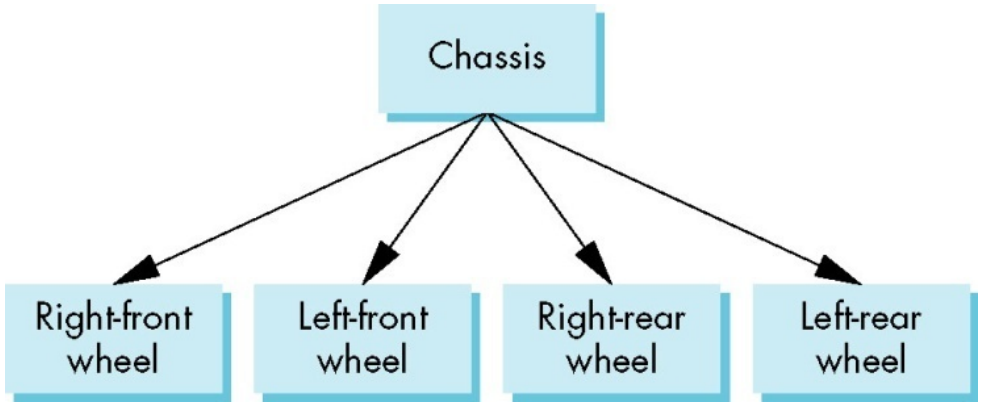


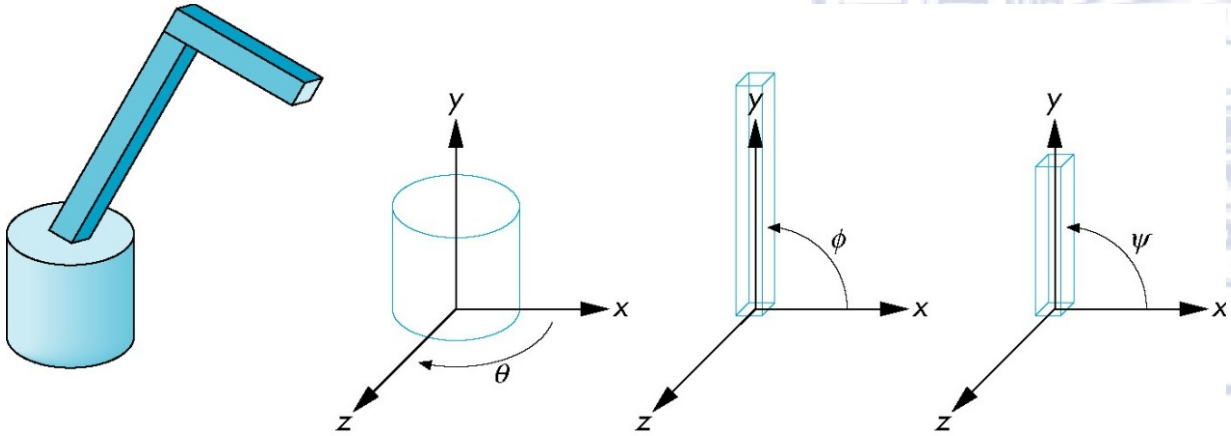



# Modellazione con alberi




- Informazione su nodi e edge
- Nodi
  - Cosa disegnare
  - Puntatori ai figli
- Edges
  - Potrebbero memorizzare trasformazioni geometriche incrementali

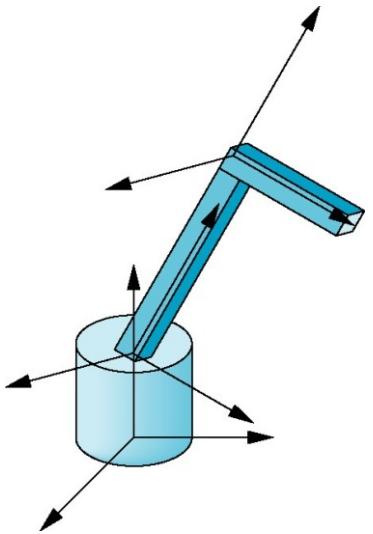
# Es. Braccio robotico (da Angel)




robot arm

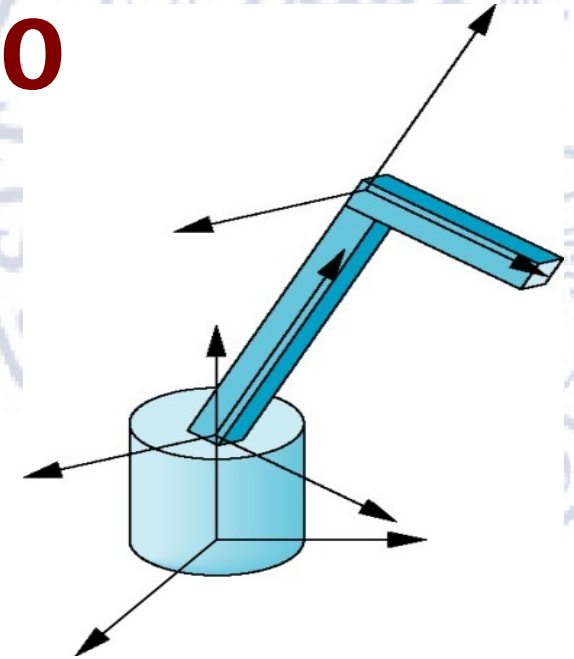
parts in their own coodinate systems

- Parti connesse alle giunture
- Stato definito dagli angoli tra i bracci alle giunture



# Braccio robotico

- La base ruota (angolo)
- Braccio inferiore attaccato alla base
  - Posizione dipende da rotazione della base
  - Può anche traslare e ruotare rispetto alla base alla giuntura
- Braccio superiore attaccato all'inferiore
  - Posizione dipende da base e braccio inferiore
  - Ulteriore traslazione e rotazione



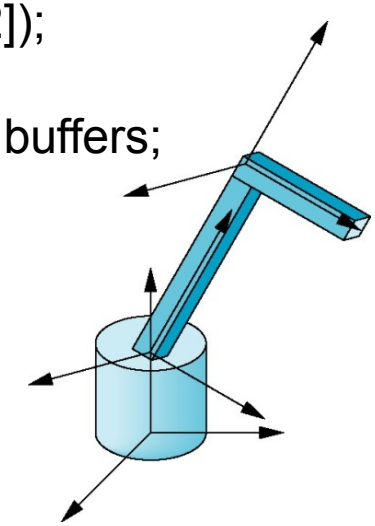
- Rotation of base: $\mathbf{R}_b$
  - Apply $\mathbf{M} = \mathbf{R}_b$ to base
- Translate lower arm relative to base: $\mathbf{T}_{lu}$
- Rotate lower arm around joint: $\mathbf{R}_{lu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ to lower arm
- Translate upper arm relative to upper arm: $\mathbf{T}_{uu}$
- Rotate upper arm around joint: $\mathbf{R}_{uu}$
  - Apply $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$ to upper arm

# Disegnare

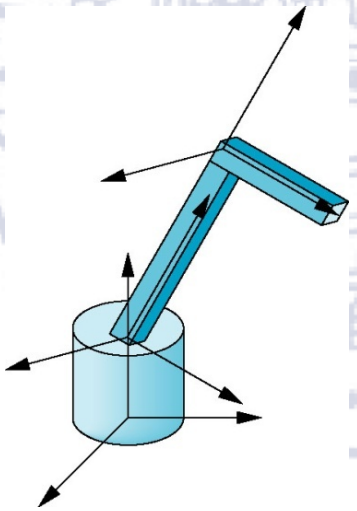
- Rotazione della base: $\mathbf{R}_b$
  - Applico $\mathbf{M} = \mathbf{R}_b$ alla base
- Traslazione rispetto base $\mathbf{T}_{lu}$
- Rotazione braccio infe: $\mathbf{R}_{lu}$
  - Applico $\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu}$ al braccio inf
- Traslazione relativa braccio superiore: $\mathbf{T}_{uu}$
- Rotazione sul giunto $\mathbf{R}_{uu}$
  - Trasformazione globale braccio superiore

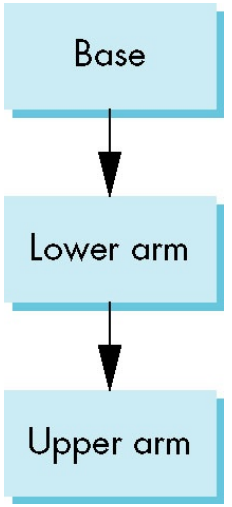
$$\mathbf{M} = \mathbf{R}_b \mathbf{T}_{lu} \mathbf{R}_{lu} \mathbf{T}_{uu} \mathbf{R}_{uu}$$

```
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
model_view = RotateY(theta[0]);
base();
model_view =model_view*Translate(0.0,
    BASE_HEIGHT, 0.0)
*RotateZ(theta[1]);
lower_arm();
model_view = model_view*Translate(0.0,
    LOWER_ARM_HEIGHT, 0.0)
*RotateZ(theta[2]);
upper_arm();
// funz per swap buffers;
}
```



```
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
model_view = RotateY(theta[0]);
base();
model_view = model_view*Translate(0.0,
   BASE_HEIGHT, 0.0)
*RotateZ(theta[1]);
lower_arm();
model_view = model_view*Translate(0.0,
   LOWER_ARM_HEIGHT, 0.0)
*RotateZ(theta[2]);
upper_arm();
// qui funzione per swap buffers
}
```

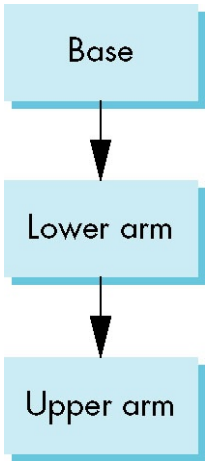




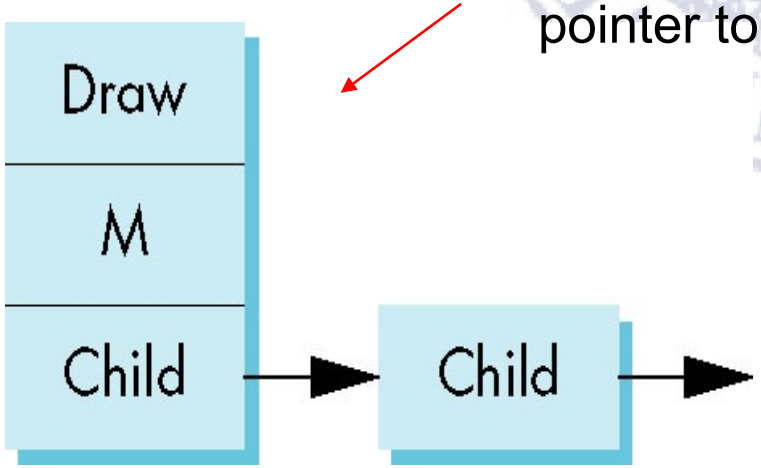

- Il codice mostra le relazioni tra le parti del modello
- Si possono cambiare i parametri dell'aspetto senza toccare la postura
- Generalizzabile

# Struttura nodi

```
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
model_view = RotateY(theta[0]);
base();
model_view = model_view*Translate(0.0,
     BASE_HEIGHT, 0.0)
*RotateZ(theta[1]);
lower_arm();
model_view = model_view*Translate(0.0,
     LOWER_ARM_HEIGHT, 0.0)
*RotateZ(theta[2]);
upper_arm();
// qui uso funzione libreria per swap buffer
}
```




Code for drawing part or pointer to drawing function



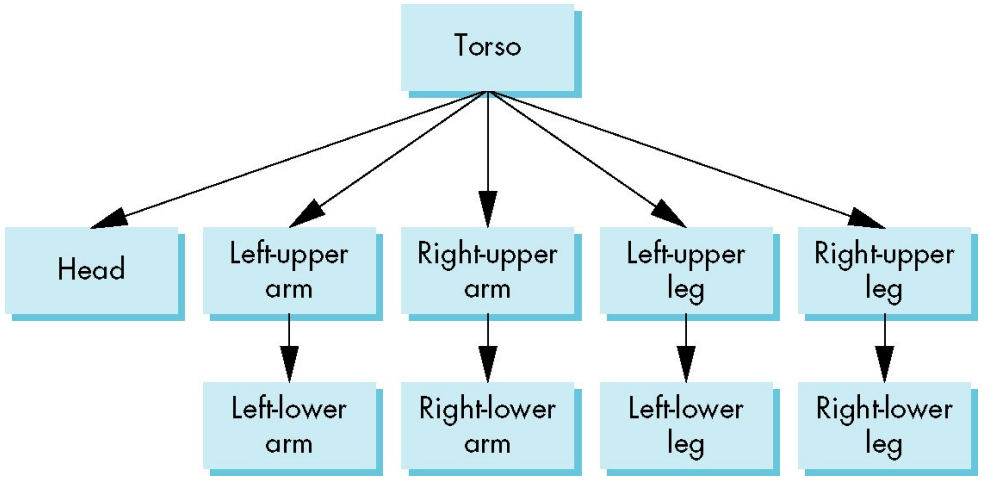

linked list of pointers to children

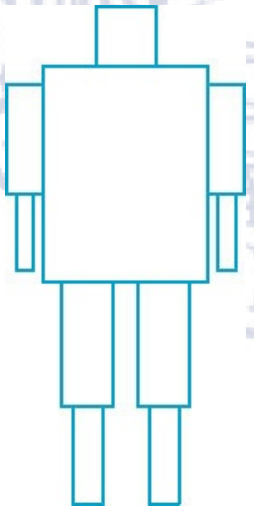
matrix relating node to parent

# In generale

- Occorre gestire eredità multiple
  - Un nodo può avere più figli
- Sarà necessario gestire modifiche a runtime della struttura per animazione
  - Variare parametri
  - Creare o distruggere nodi

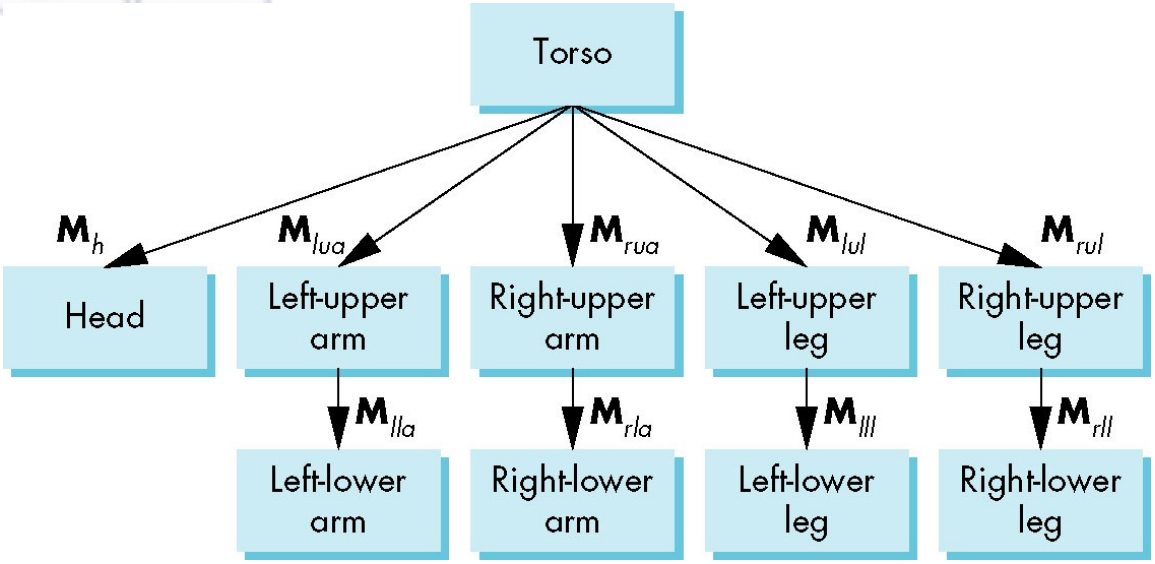
# Costruire un modello

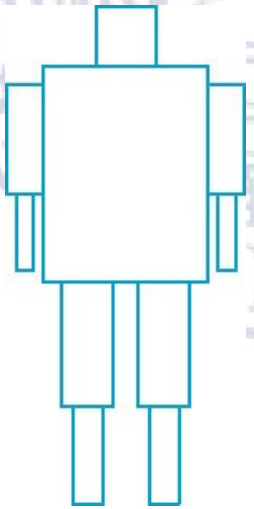


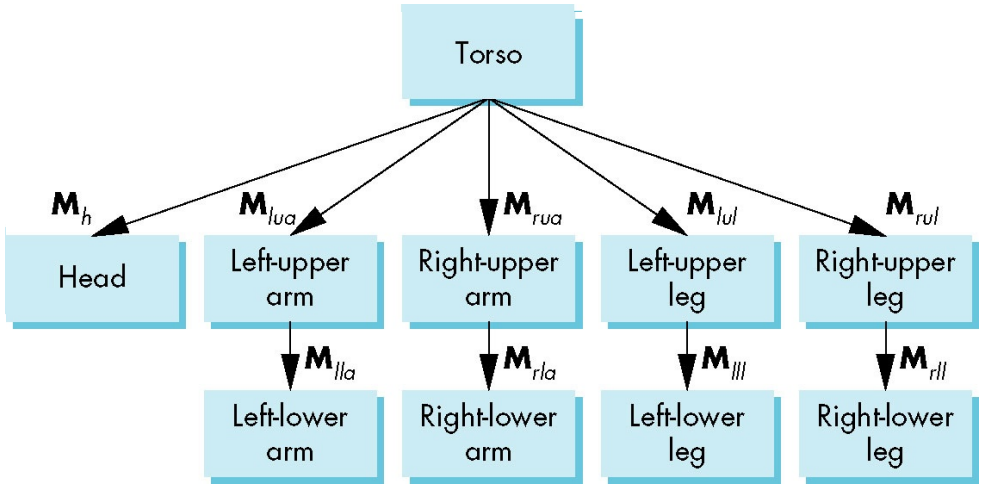
- Supponiamo di creare facilmente la geometria delle parti (es. cilindri)
- Vogliamo creare strutture per accedere alle parti
  - torso(), left_upper_arm()
- Dentro matrici che descrivono posizione del nodo rispetto al genitore

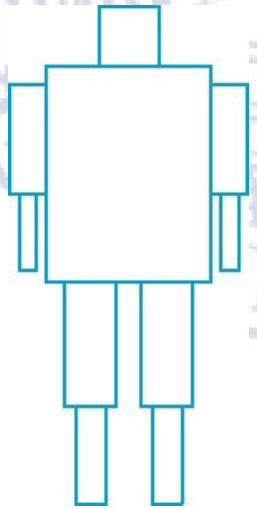
# Costruire un modello



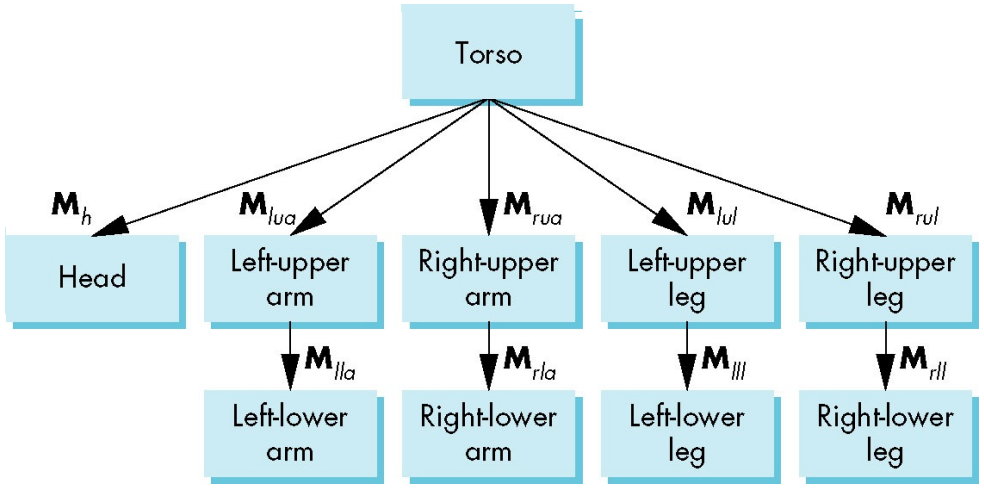



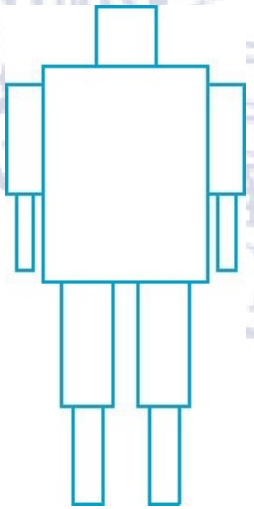
- Supponiamo di creare facilmente la geometria delle parti (es. cilindri)
- Vogliamo creare strutture per accedere alle parti
  - torso(), left_upper_arm()
- Dentro matrici che descrivono posizione del nodo rispetto al genitore



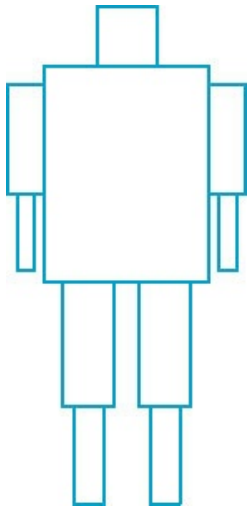
- La posa è determinata da 11 angoli alle giunture (due per la testa e uno per le altre connessioni)
- Fare il rendering si riduce all'attraversamento di un grafo
  - Visito ogni nodo
  - Eseguo la Display function a ognuno di essi: questa descrive la parte del corpo associata, applica la corretta trasformazione per posizione e orientazione

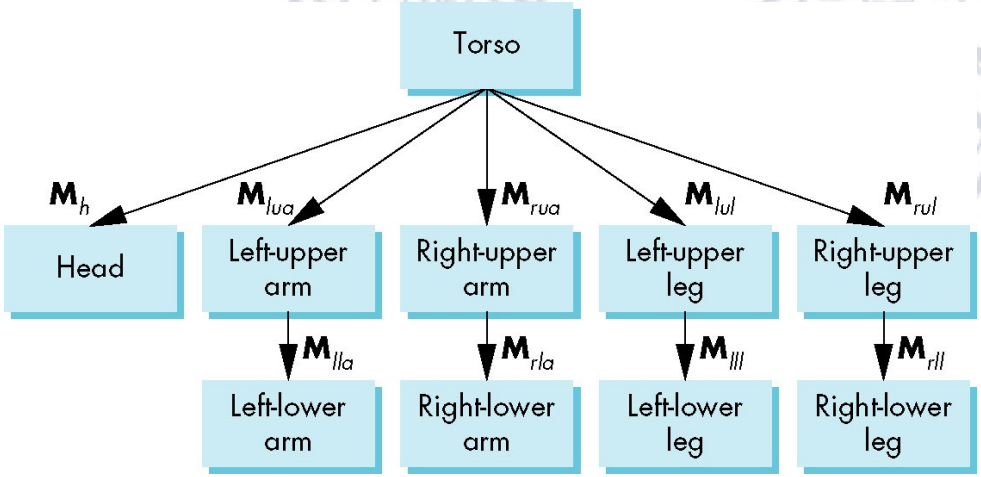


- 10 matrici rilevanti
  - **M** posiziona e orienta il orso
  - $\mathbf{M}_h$ posiziona la testa rispetto al torso
  - $\mathbf{M}_{lua}$, $\mathbf{M}_{rua}$, $\mathbf{M}_{lul}$, $\mathbf{M}_{rul}$ posizionano braccia e gambe
  - $\mathbf{M}_{lla}$, $\mathbf{M}_{rla}$, $\mathbf{M}_{lll}$, $\mathbf{M}_{rll}$ posizionano avambracci e parte inferiore delle gambe

# Visualizzazione

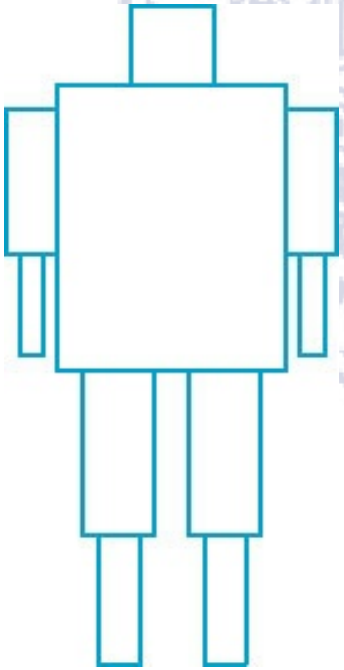


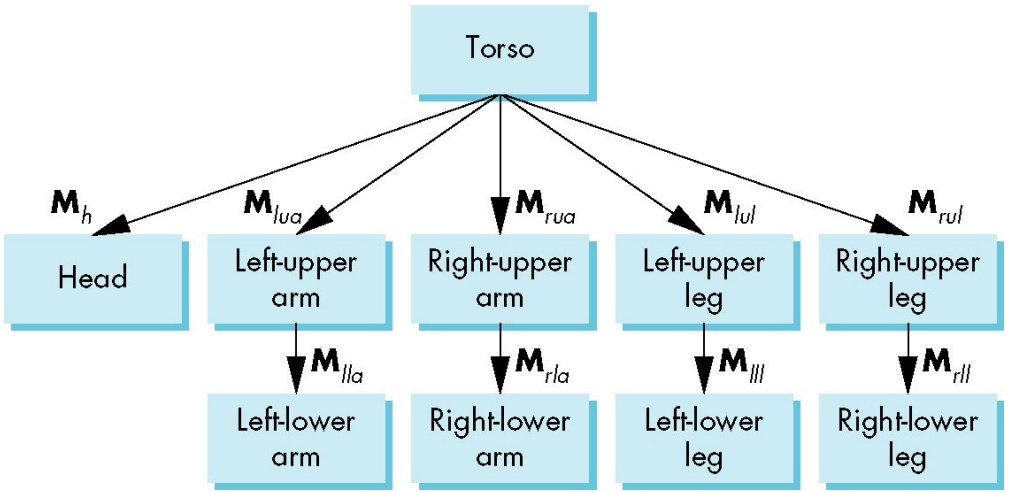



- Metto la model-view matrix a $\mathbf{M}$ e disegno torso
- Set model-view matrix to $\mathbf{MM}_h$ and draw head
- Metto la model-view matrix a $\mathbf{Mm}_{luae}$ e disegno l'arto superiore...
- Invece di ricalcolare $\mathbf{MM}_{lua}$ da zero o usare l'inversione di matrice, si usa memorizzare le matrici parziali in uno stack per poterle recuperare durante l'attraversamento

# Codifica con stack

```
mat4 model_view;
matrix_stack mvstack;
figure()
{
mvstack.push(model_view);
torso();
model_view = model_view*Translate()*Rotate();
head();
model_view = mvstack.pop();
mvstack.push(model_view);
model_view = model_view*Translate()*Rotate();
left_upper_arm();
model_view = mvstack.pop();
mvstack.push(model_view);
model_view = Translate()*Rotate();
left_lower_arm();
model_view = mvstack.pop();
mvstack.push(model_view);
model_view = Translate()*Rotate();
right_upper_arm();
model_view = mvstack.pop();
mvstack.push(model_view);
.
.
.
}
```





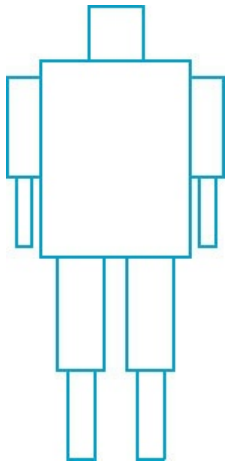


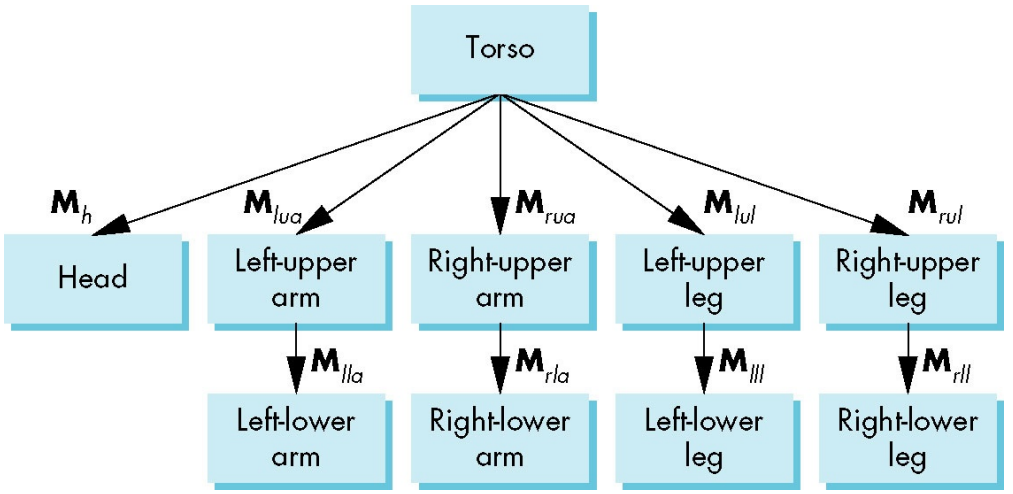
- Codice per una parte del corpo

```
void torso()
{
mvstack.push(model_view);
instance = Translate(0.0, 0.5*TORSO_HEIGHT,0.0)
*Scale(TORSO_WIDTH, TORSO_HEIGHT, TORSO_WIDTH);
glUniformMatrix4fv(model_view_loc, 16, GL_TRUE,
model_view*instance);
colorcube();
glDrawArrays(GL_TRIANGLES, 0, N);
model_view = mvstack.pop();
}
```

# Attraversamento

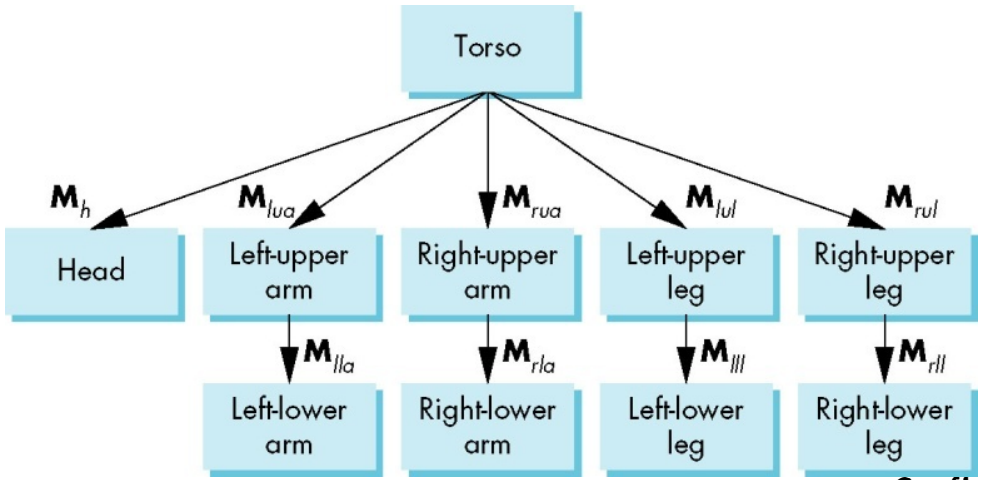
```
void traverse(treenode* root)
{
if (root == NULL) return;
mvstack.push(model_view);
model_view = model_view*root->m;
root->f();
if (root->child != NULL) traverse(root->child);
model_view = mvstack.pop();
if (root->sibling != NULL) traverse(root->sibling);
}
```





# Note

```
figure() {
   glPushMatrix()
   torso();
   glRotate3f(…);
   head();
   glPopMatrix();
   glPushMatrix();
   glTranslate3f(…);
   glRotate3f(…);
   left_upper_arm();
   glPopMatrix();
   glPushMatrix();
```



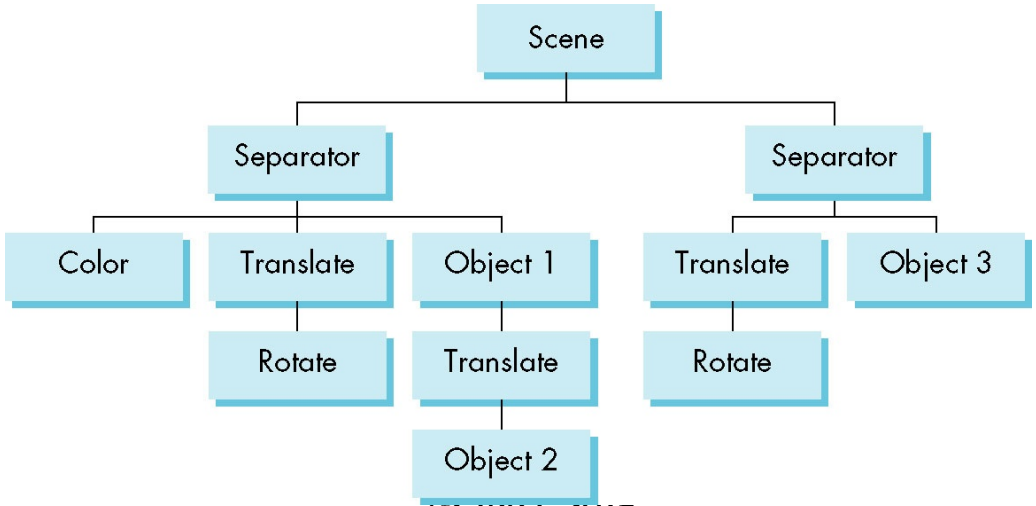

- Occorre salvare la model-view matrix prima di moltiplicarla per la matrice del nodo
  - La matrice aggiornata la usano i figli ma non i fratelli (siblings) che hanno la loro
- Il programma di attraversamento è generico per strutture di questo tipo ( left-child right-sibling tree)
- L'ordine di attraversamento può cambiare le cose a causa dei cambiamenti di stato del sistema

# Nota

- OpenGL non è orientato agli oggetti
- Consideriamo un oggetto di quelli che istanziamo, es. una sfera verde
  - Abbiamo il modello poligonale
  - Ma il colore dipende dallo stato di OpenGLnon è una proprietà dell'oggetto
- Di solito si gestiscono gli “oggetti” con la struttura sovrastante del linguaggio  object-oriented
  - Ma naturalmente non ce ne occupuiamo qui

# Scene graph

- Possiamo rappresentare I modelli nelle scene con alberi o DAG
- Possiamo attraversare le strutture per fare rendering
  - I nodi definiscono trasformazioni e funzioni di visualizzazione
- Possiamo anche rappresentare tutti gli elementi (camere, luci materiali oltre alle geometrie) come oggetti rappresentati in un albero e fare il rendering con l'attraversamento
  - Scene graph



# Implementazione in API/Toolkit

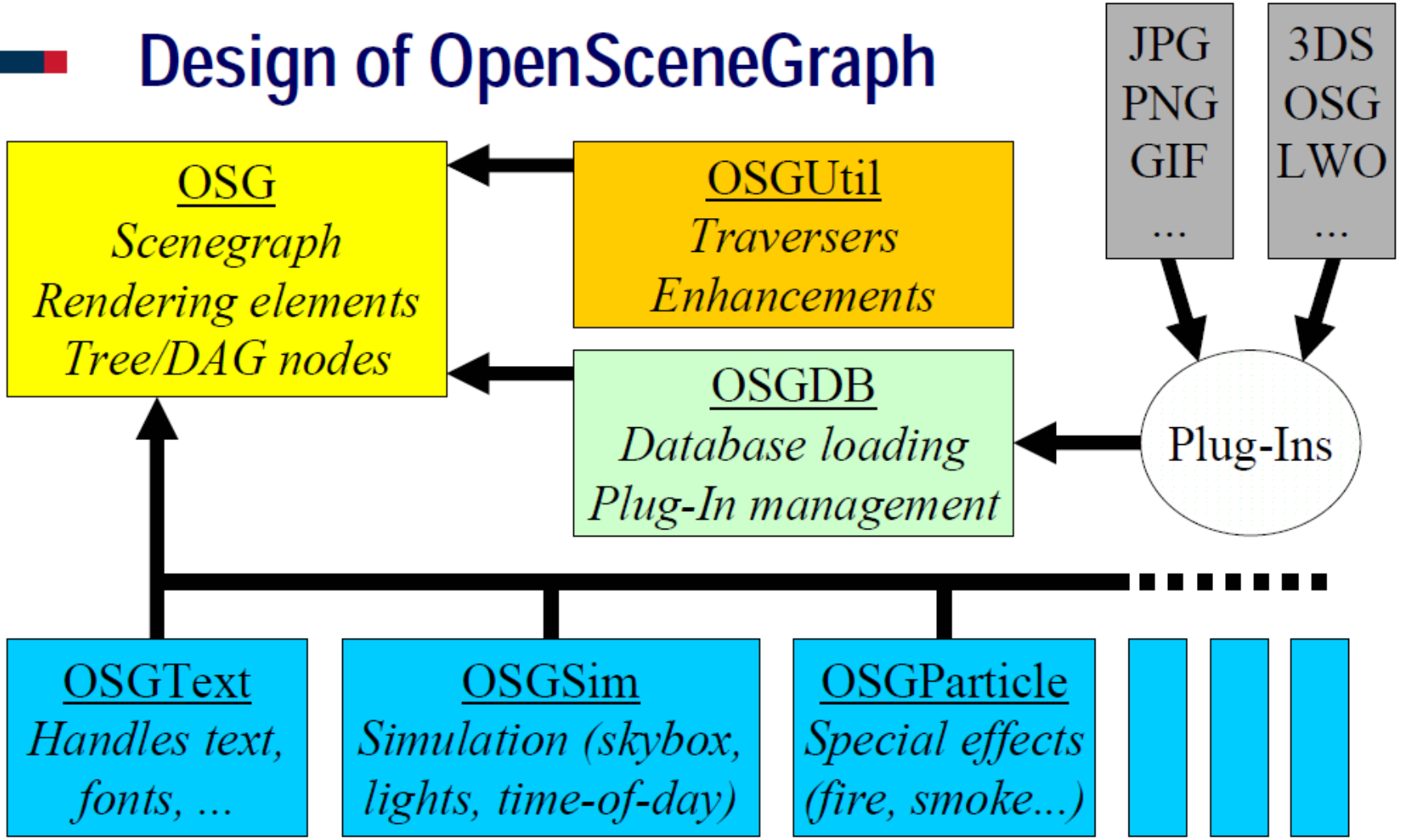
- Open Scene Graph (OSG),
  - Cross-platform, supports culling, sorting, level of detail
- Inventor, Java3D,
- Ogre3D (http://www.ogre3d.org/)
  - Games, high-performance rendering
- Nvidia scene graph (NVSG), VRML,X3D
- Scene graphs can also be described by a file (text or binary)
  - Implementation independent way of transporting scenes
  - Supported by scene graph APIs
- However, primitives supported should match capabilities of graphics systems
  - Hence most scene graph APIs are built on top of OpenGL or DirectX (for PCs)

# OSG


IVE
Modeller
Skærm on/off
Wireframe
Viewpoints
Exit
DR
Dette er den udvendige model af DR byen. På knappen 'Viewpoints' i det øverste hjørne kan du vælge position i modellen. Knappen 'Wireframe' viser det tekniske skelet, modellen er bygget af.
Mouse
Joystick
Cluster
Model
Navigator
Viewpoint Statistics
Materials: 937
Polygons: 90007
Vertices: 270021
Fps: 0Hz
Showing model: DR Byen

# OSG

## Design of OpenSceneGraph



# Java 3D

- Advantages
- Open source cross-platform development.
- Low-level control of scene graph and objects.
- Can be used with other Java and native libraries.

- Disatvantages
- Scene manipulation done strictly through source, leads to slow turnaround.
- Higher level control is up to the programmer.
- 3D sound very buggy.
- Community support only, no longer any commercial support.

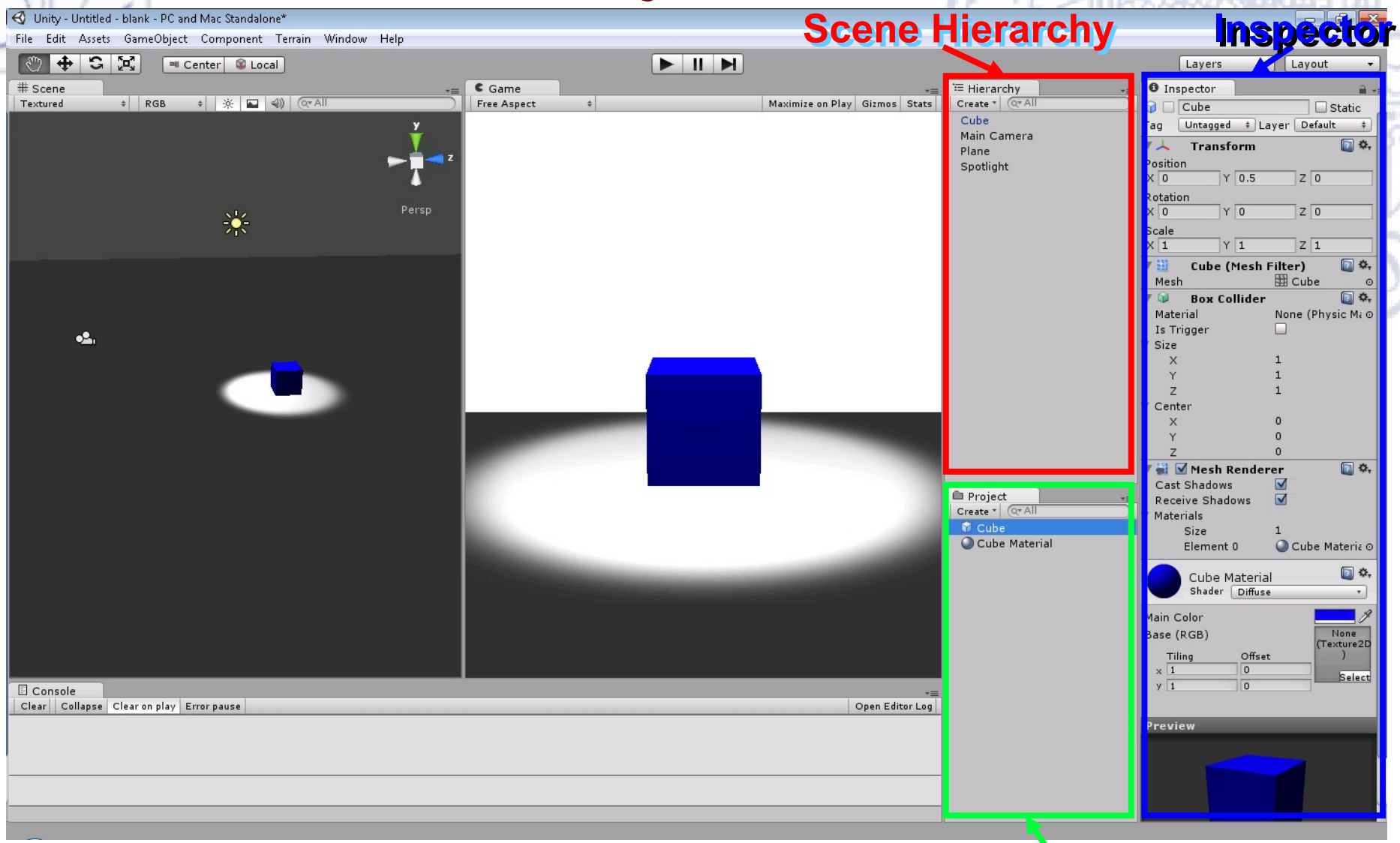
# Game engines: Unity3D

- Free edition offers robust development environment and educational licenses available.
- Supports multiple programming languages to design and manipulate the scene.
- External library and .Net support allows seamless communication with additional hardware devices.
- Easy-to-use graphical interface allows live scene editing for efficient development and testing.
- Quick turnaround times.
- Works on almost all available platforms.

# Unity - Physics Engine

- Unity uses NVIDIA's PhysX Engine.
- Streamlined physics modeling for rigid bodies, cars, character ragdolls, soft bodies, joints, and cloths.
- By simply attaching a rigid body to a game object and adding forces, realistic physical interactions can be created.
- Objects with rigid bodies attached will interact with each other.
- Colliders are used to control these object interactions and trigger collision events.

# Unity - GUI


Scene Hierarchy
Inspector
Project Panel
Unity - Untitled - blank - PC and Mac Standalone*
File Edit Assets GameObject Component Terrain Window Help
Center Local
Layers Layout
Scene
Textured RGB
Persp
Game
Free Aspect Maximize on Play Gizmos Stats
Hierarchy
Create
Cube
Main Camera
Plane
Spotlight
Inspector
Cube Static
Tag Untagged Layer Default
Transform
Position X 0 Y 0.5 Z 0
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1
Cube (Mesh Filter)
Mesh Cube
Box Collider
Material None (Physic M
Is Trigger
Size X 1 Y 1 Z 1
Center X 0 Y 0 Z 0
Mesh Renderer
Cast Shadows
Receive Shadows
Materials
Size 1
Element 0 Cube Materia
Cube Material
Shader Diffuse
Main Color
Base (RGB)
None (Texture2D)
Tiling Offset
x 1 0
y 1 0
Select
Preview
Project
Create
Cube
Cube Material
Console
Clear Collapse Clear on play Error pause Open Editor Log

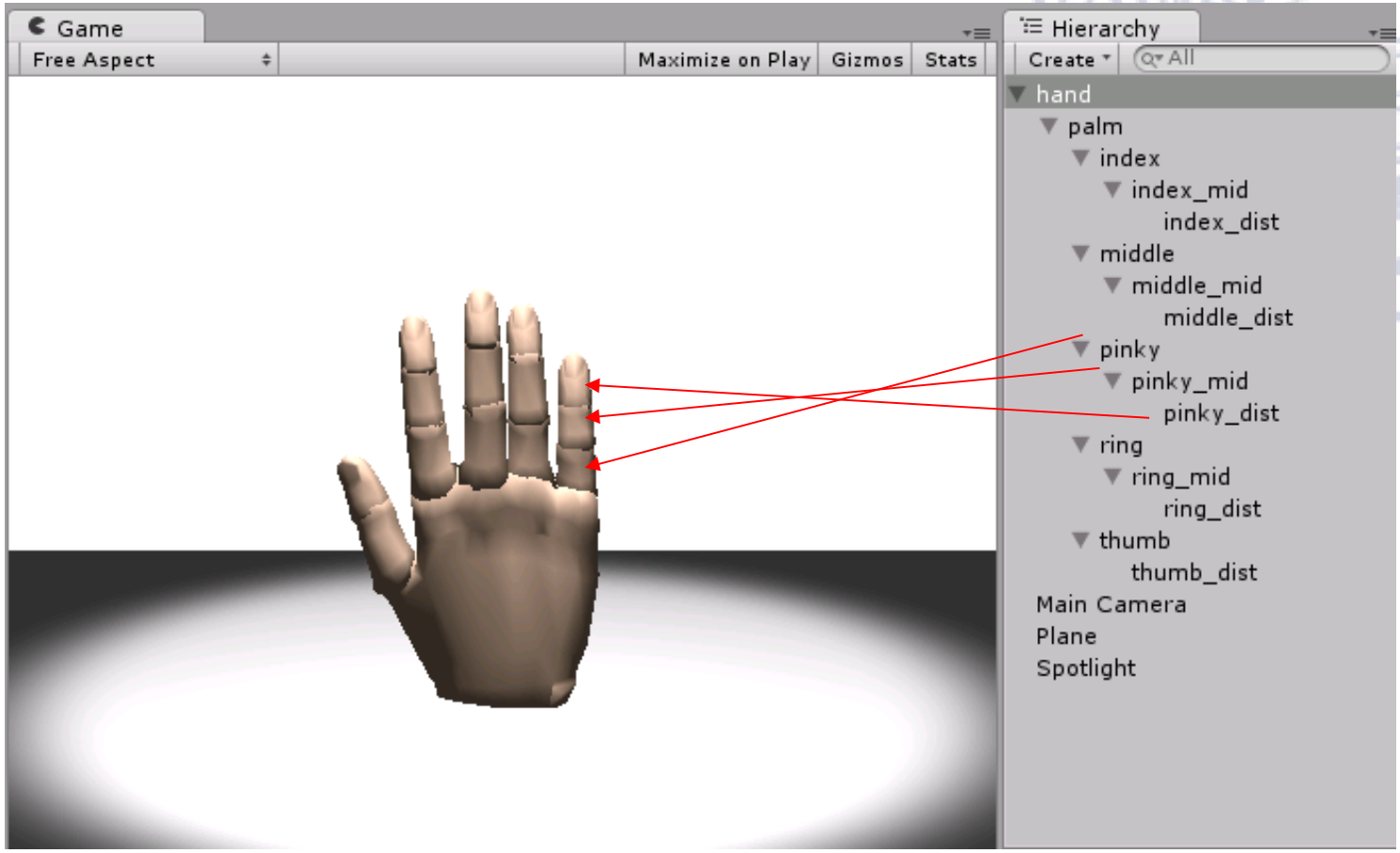
# Unity – Project Panel

- This panel shows all of the available game assets in the current project directory.
- Game assets can include scripts, prefabs, 3D models, texture images, sound files, fonts, etc...
- New assets can be added by simply dragging them into the project panel or placing them into the project directory.
- These files can be edited and saved using external programs and the scene will be updated automatically.
- Unity supports a number of 3D model formats and converts to the Autodesk FBX format when added to the project.

# Unity - Scene Hierarchy

- Provides a means of adding new game objects and managing existing ones.
- Game objects can contain a combination of transforms, graphics objects, physics colliders, materials, sounds, lights, and scripts.
- Each game object in the hierarchal tree represents a node in the scene graph.
- Similarly to VRML and Java3D, the scene graph represents the relative spatial relationship of objects. Example: A finger connected to a hand will translate accordingly when the hand is moved.

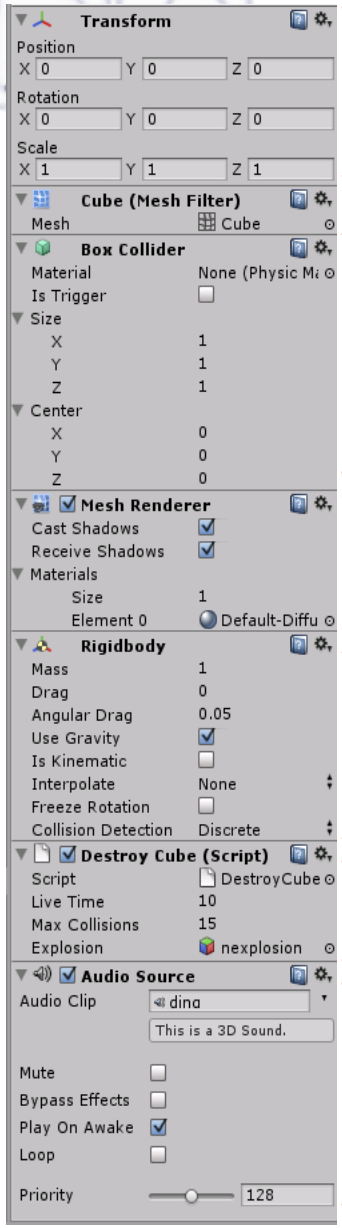
# Unity - Simple Hierarchy Example



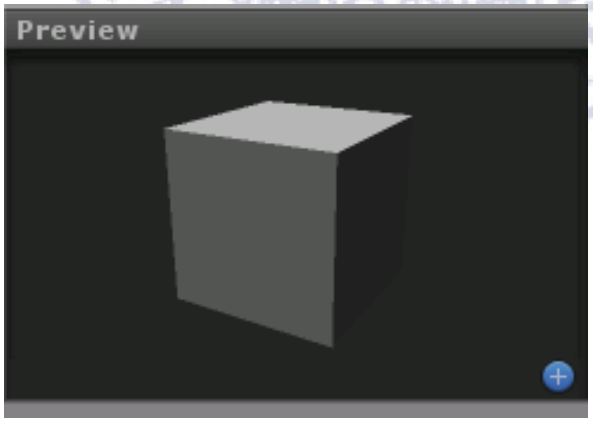
# Unity - Inspector

- Shows the components attached to currently selected game object and their properties.
- Manual control over an object's transform allows precise placement of objects.
- Variables exposed by scripts attached to the object can be viewed and set through this panel, allowing parameters to be changed without the need to edit source.
- These changes can be done while the project is live and the scene will update immediately.

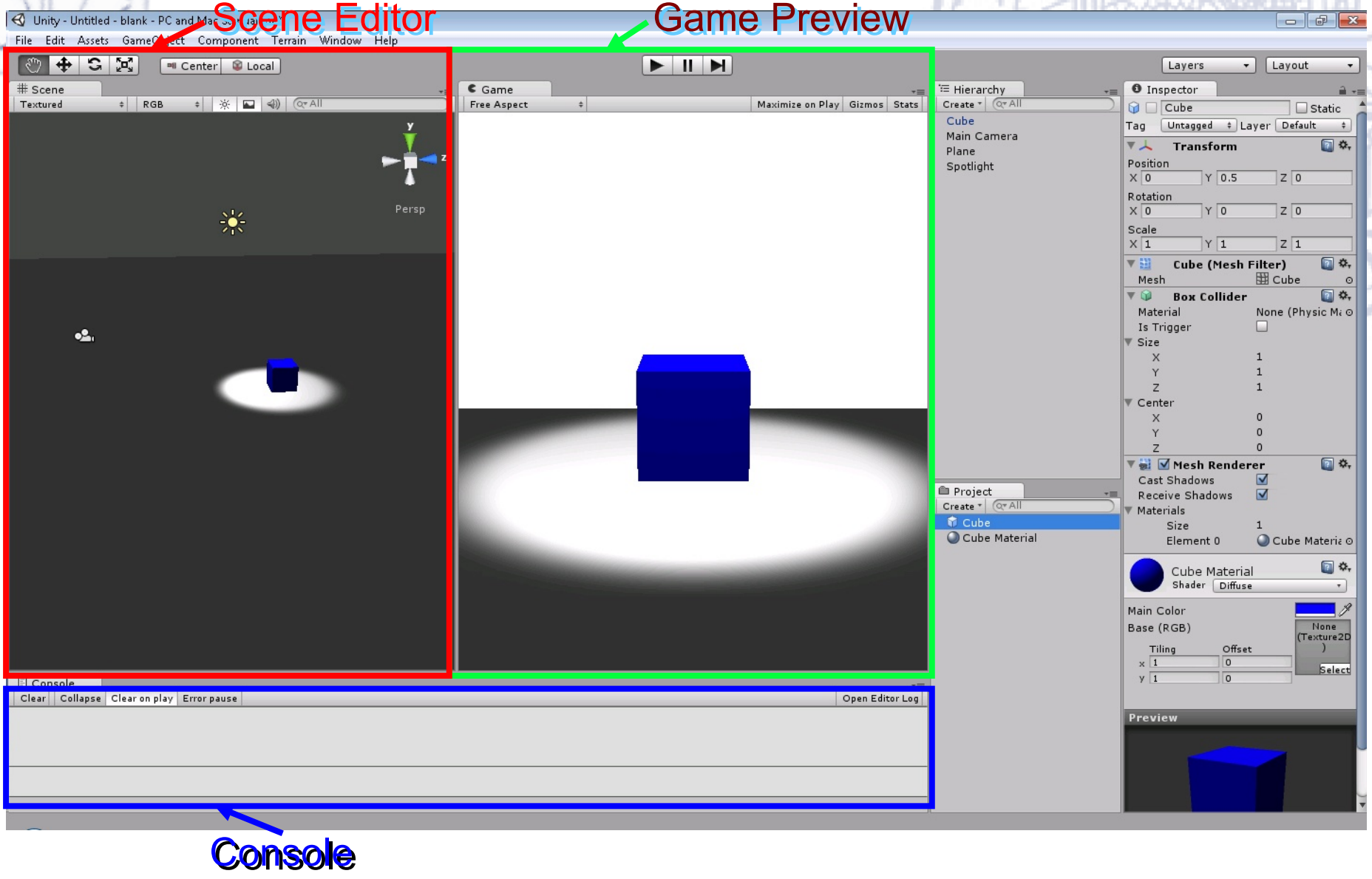
# Unity - Simple Game Object




- Defines spatial properties (Transformation matrix)
- Controls physics and physical interactions.
- Graphics mesh, this is what you will actually see.
- Script to destroy object after N collisions or after elapsed time. Contains particle emitter for explosion effect.
- Sound associated with this object.



# Unity - GUI


Scene Editor
Game Preview
Console
File Edit Assets GameObject Component Terrain Window Help
Center
Local
Scene
Textured
RGB
Persp
Game
Free Aspect
Maximize on Play
Gizmos
Stats
Hierarchy
Create
Cube
Main Camera
Plane
Spotlight
Project
Create
Cube
Cube Material
Inspector
Cube
Static
Tag
Untagged
Layer
Default
Transform
Position
Rotation
Scale
Cube (Mesh Filter)
Mesh
Box Collider
Material
Is Trigger
Size
Center
Mesh Renderer
Cast Shadows
Receive Shadows
Materials
Element 0
Cube Material
Shader
Diffuse
Main Color
Base (RGB)
Tiling
Offset
Select
Preview
Console
Clear
Collapse
Clear on play
Error pause
Open Editor Log

# Unity - Scene Editor

- Allows graphical monitoring and manipulation of scene objects.
- Switch between various orthogonal and perspective views.
- Objects can be translated, rotated, and scaled graphically with the mouse.
- When live, the editor works like a sandbox in which you can play around with objects without actually modifying the scene.
- Shows “Gizmos” for invisible scene objects, such as light sources, colliders, cameras, and sound effects.

# Riferimenti



- Angel (6.ed.) capitolo 8

# Domande di verifica

- Qual è la differenza tra DAG è albero
- Qual è la differenza tra oggetto logico e istanza?
- Cosa si intende per “scene graph”?
- Quale problema sussiste nell'adattare dinamicamente il dettaglio delle mesh in base al punto di vista,