Software Development Seminar

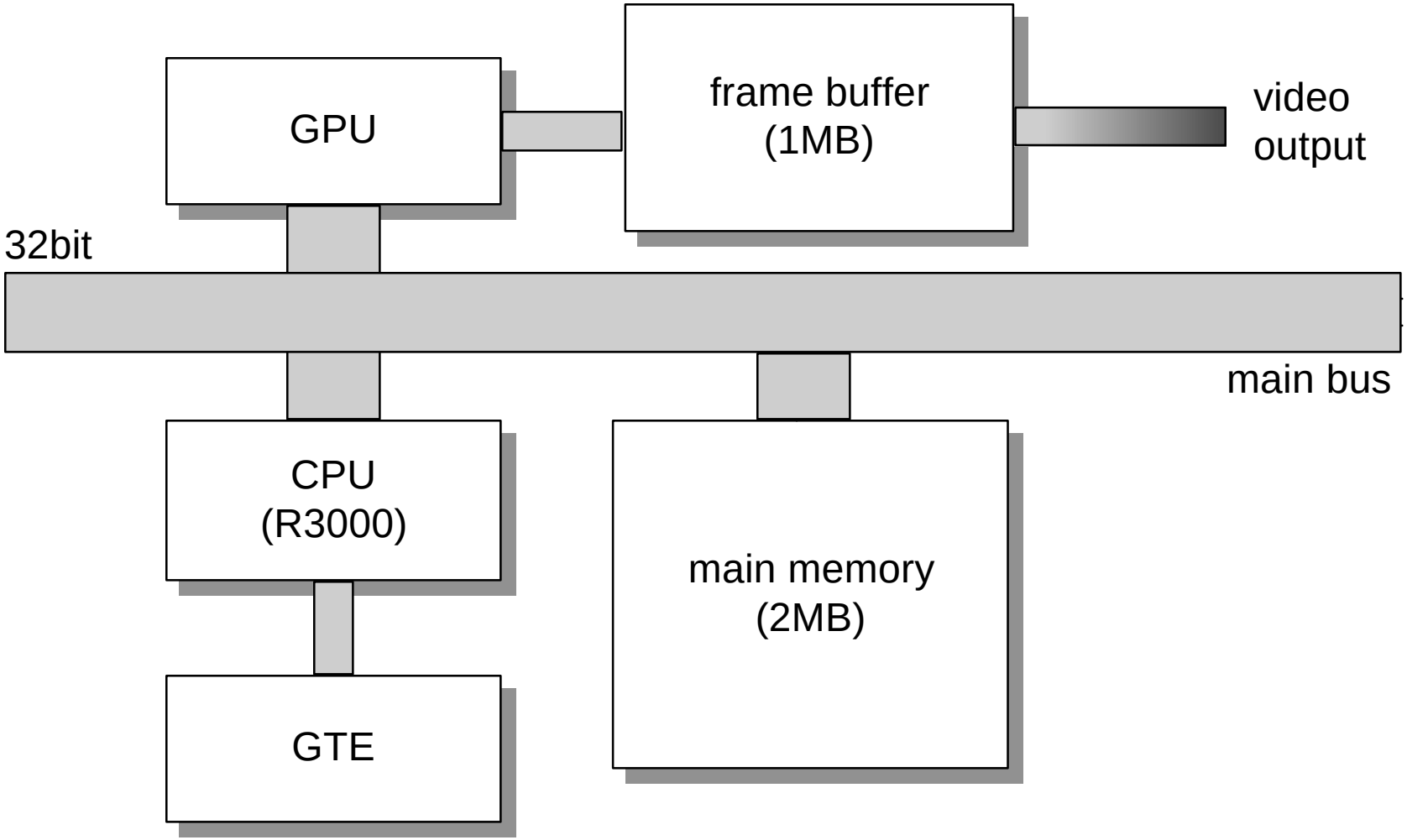
Graphics (Basic)

Overview of the Graphics System

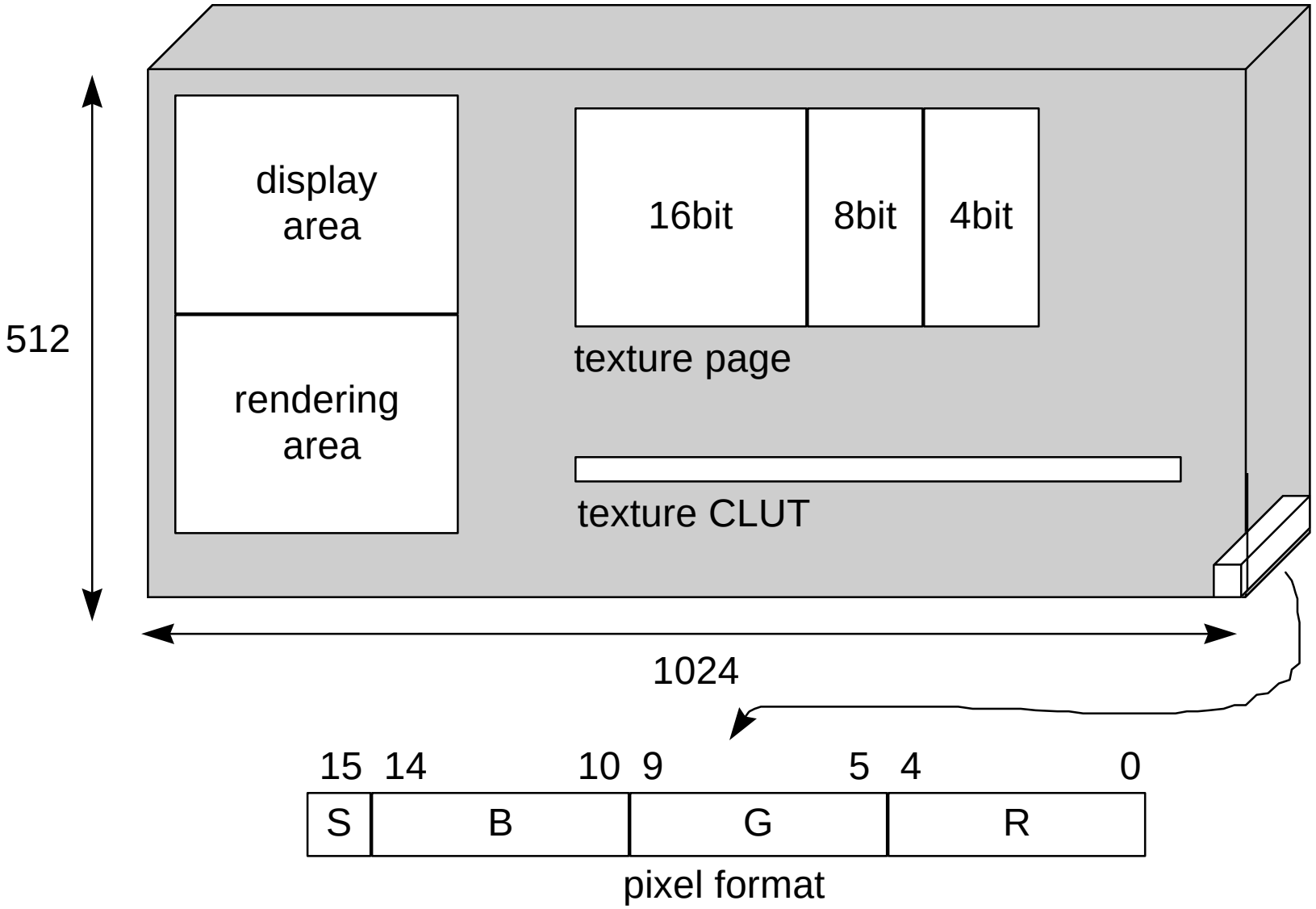
Contents

- Rendering system
  - frame buffer
  - primitives
  - ordering table (OT)
- Coordinate calculations
  - perspective transformations
  - Z sort
- Texture mapping

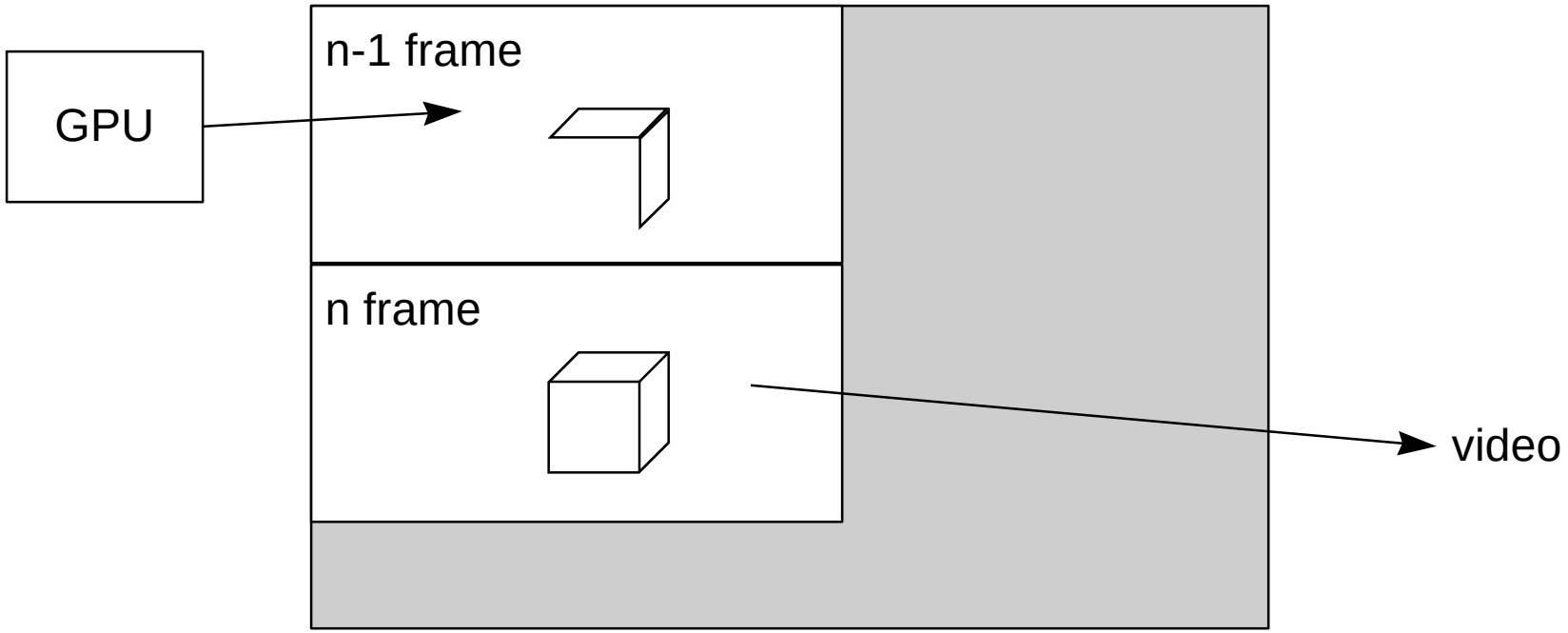
System block



Frame buffer



Frame double buffer



Separate buffers are prepared for displaying and rendering

Primitives

Basic units for rendering

texture	vertices	flat	Gouraud
none	3	POLY_F3	POLY_G3
	4	POLY_F4	POLY_G4
present	3	POLY_FT3	POLY_GT3
	4	POLY_FT4	POLY_GT4

Primitives (cont)

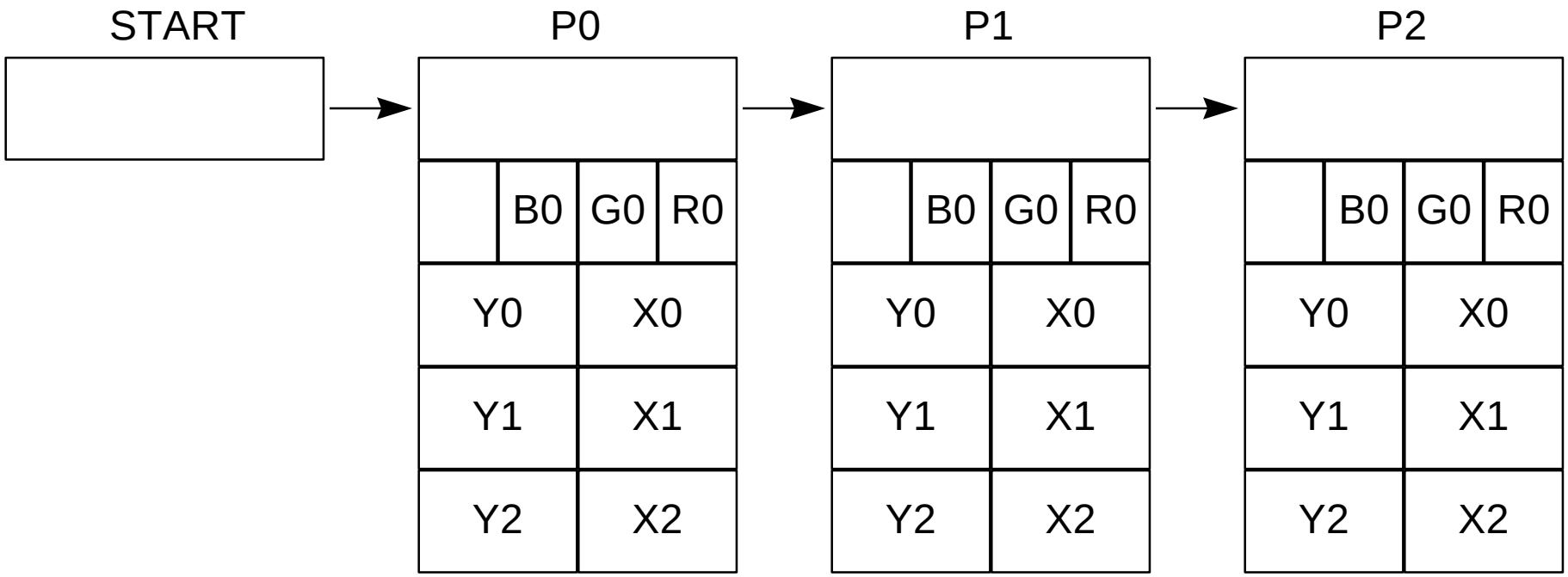
Primitive format

TAG			
CODE	B0	G0	R0
Y0		X0	
Y1		X1	
Y2		X2	

```
typedef struct {
    u_long tag;
    u_char r0, g0, b0, code;
    short x0, y0;
    short x1, y1;
    short x2, y2;
} POLY_F3;
```

TAG: pointer to next primitive

Primitive list

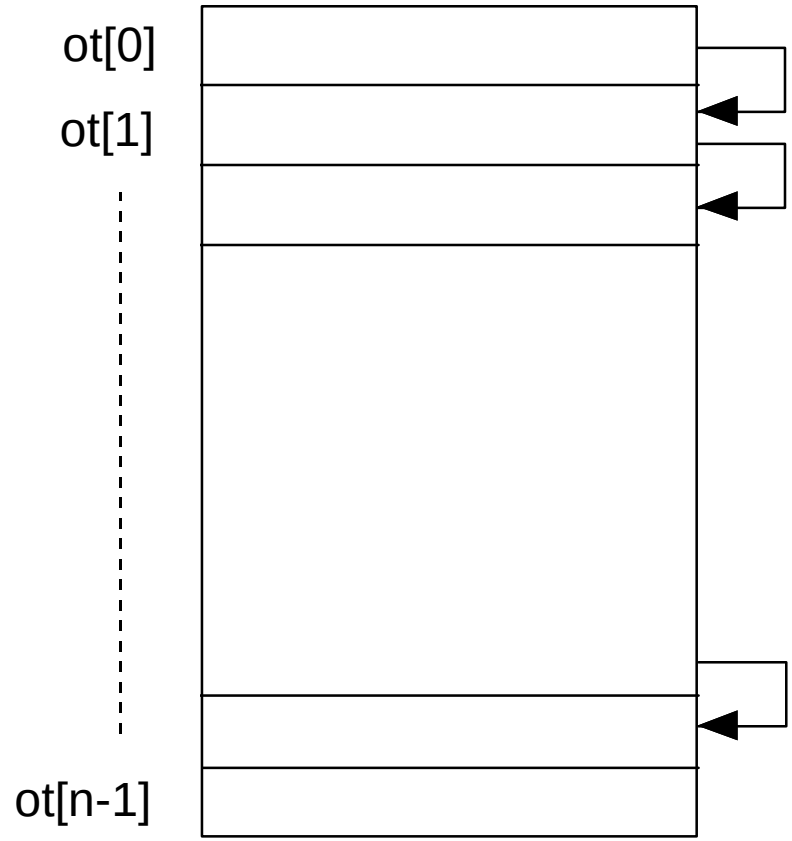


DrawOTag(&start)

rendered in the following order: → P0→ P1→ P2

Ordering table

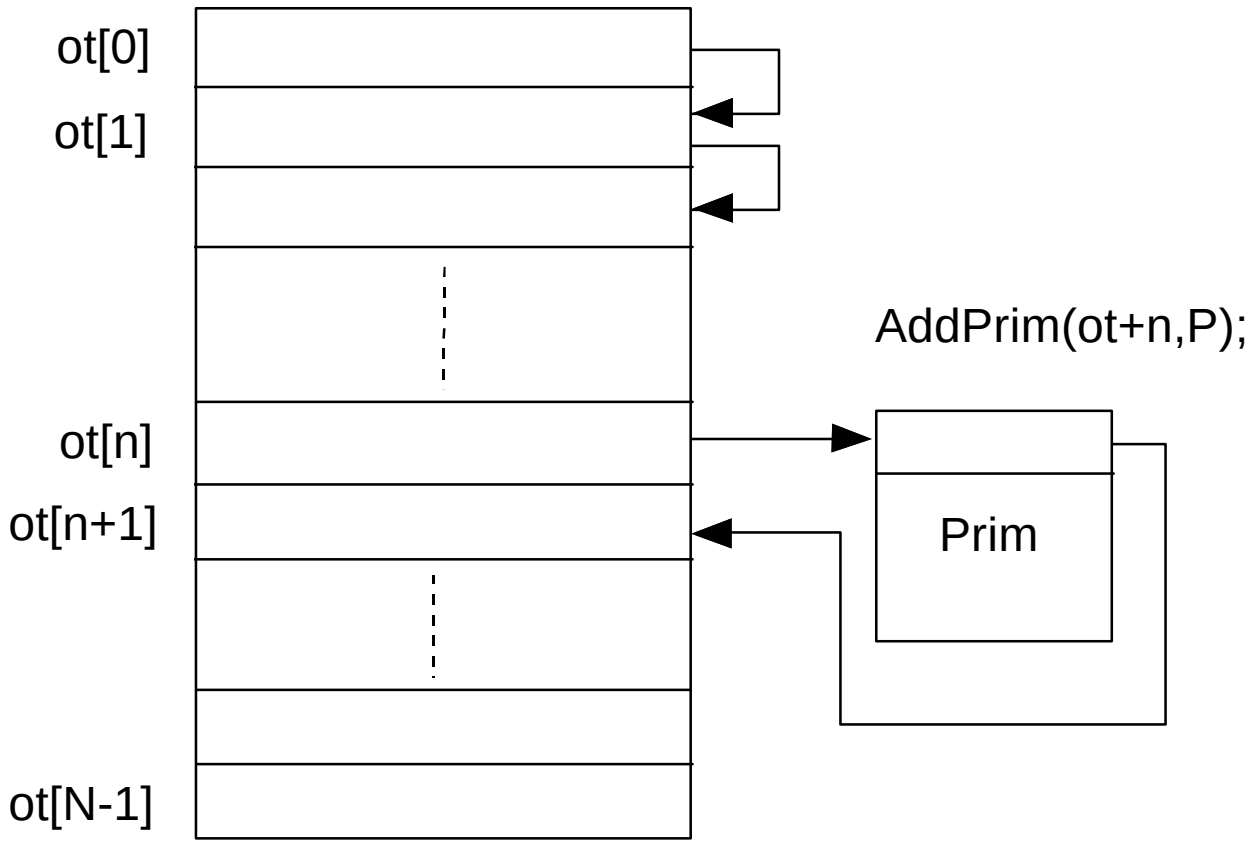
array of null primitives (primitives that only contain the TAG)



u_long ot[N];

ClearOTag(ot,N);

Link to ordering table



3D display

[1] Coordinate transforms

Rotate coordinates (Rot)
Translate coordinates (Trans)

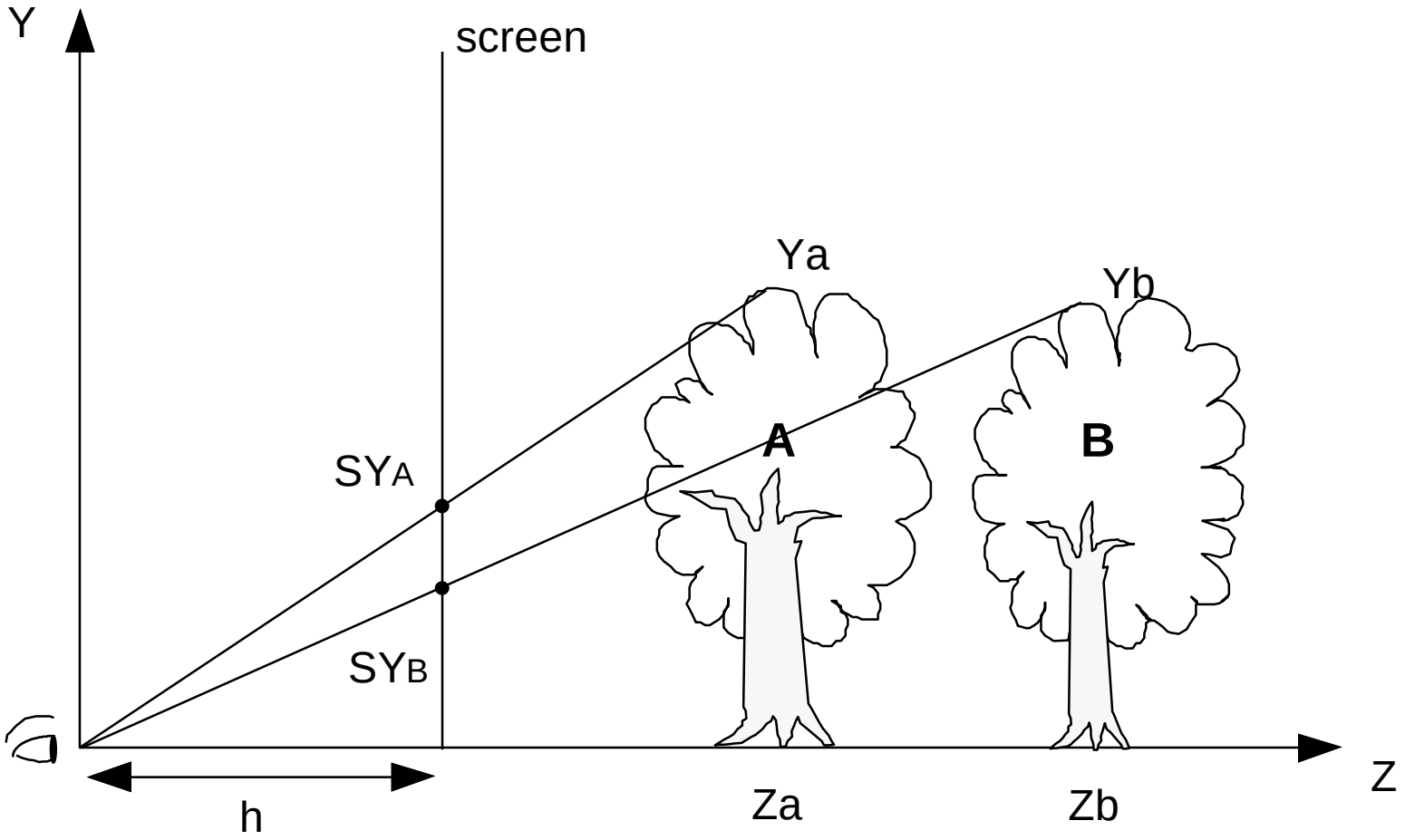
[2] Perspective transforms

Far objects are small, near objects are large (Pers)

[3] Z Sort

Render far objects first, near objects later

Perspective transform



$$SY = \frac{y}{z} \cdot h \quad \text{...} \quad \text{perspective transform}$$

Mathematical expression of Rot - Trans - Pers

$$(1)\quad \begin{bmatrix} Xs \\ Ys \\ Zs \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} Xw \\ Yw \\ Zw \end{bmatrix} + \begin{bmatrix} Tx \\ Ty \\ Tz \end{bmatrix}$$

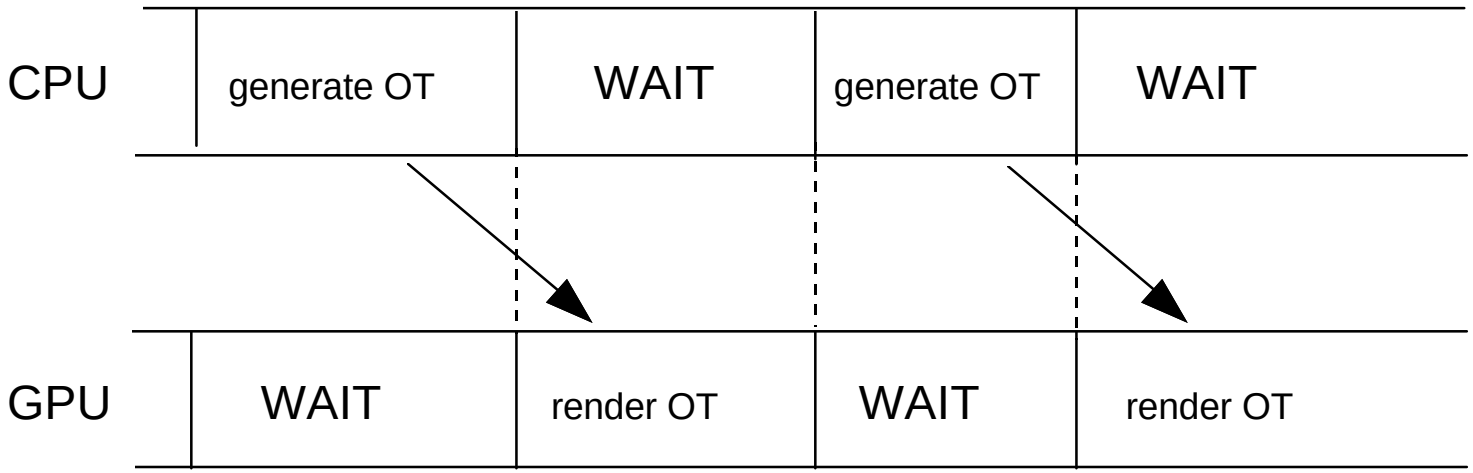
rotation matrix

translation vector

$$(2)\quad \begin{bmatrix} X \\ Y \end{bmatrix} = h \begin{bmatrix} Xs / Zs \\ Ys / Zs \end{bmatrix}$$

Primitive double buffer (1)

- simple approach

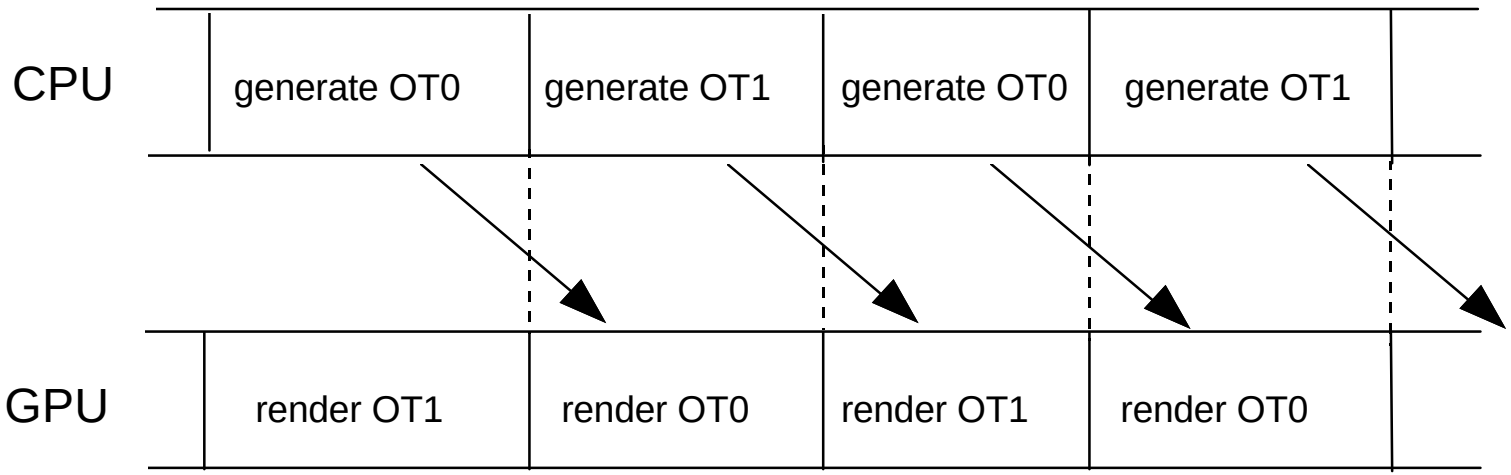


CPU and GPU cannot operate in parallel

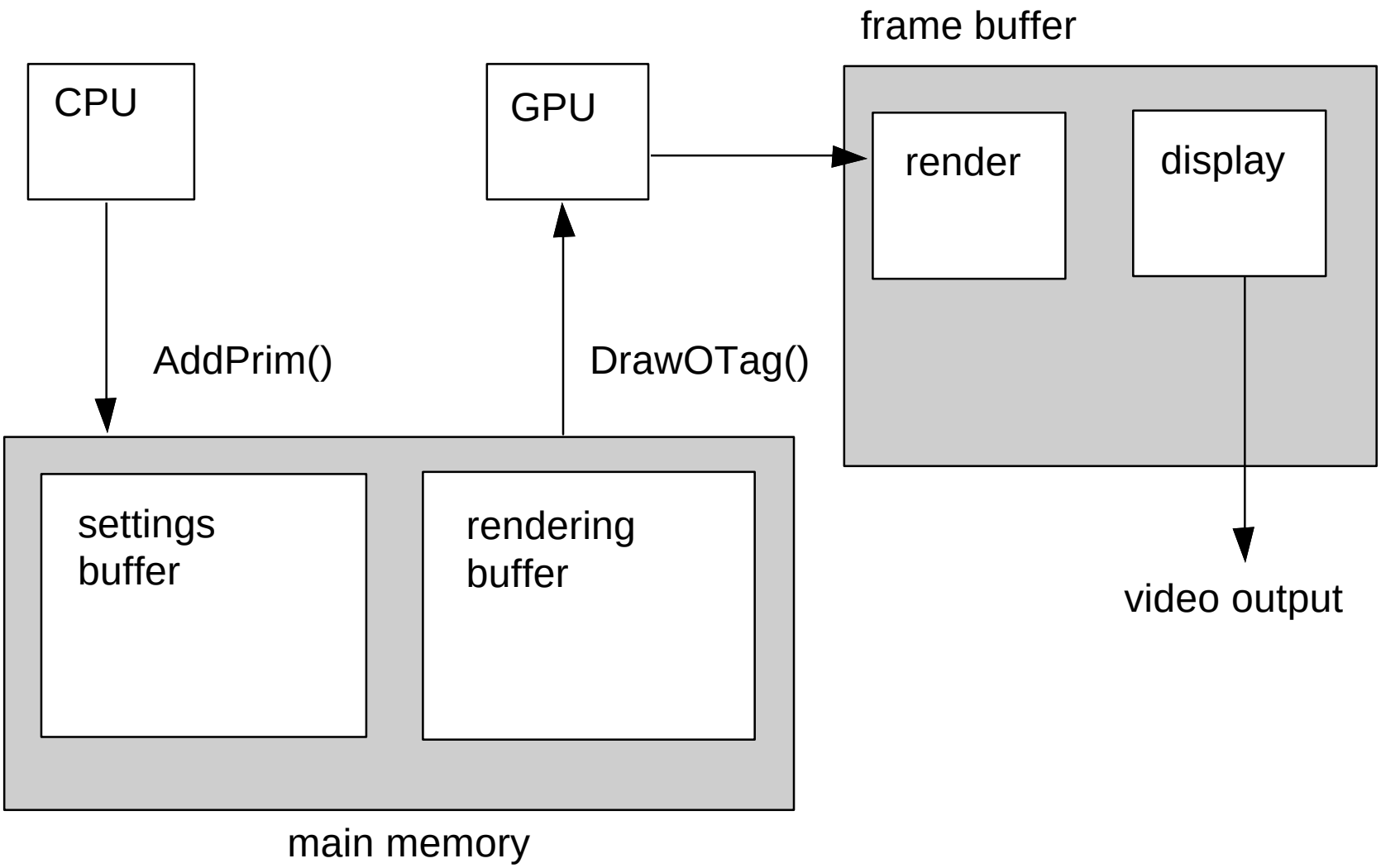
Primitive double buffer (2)

Rendering primitive buffer
Displaying primitive buffer
→ 2 sets are prepared

- CPU and GPU can operate in parallel

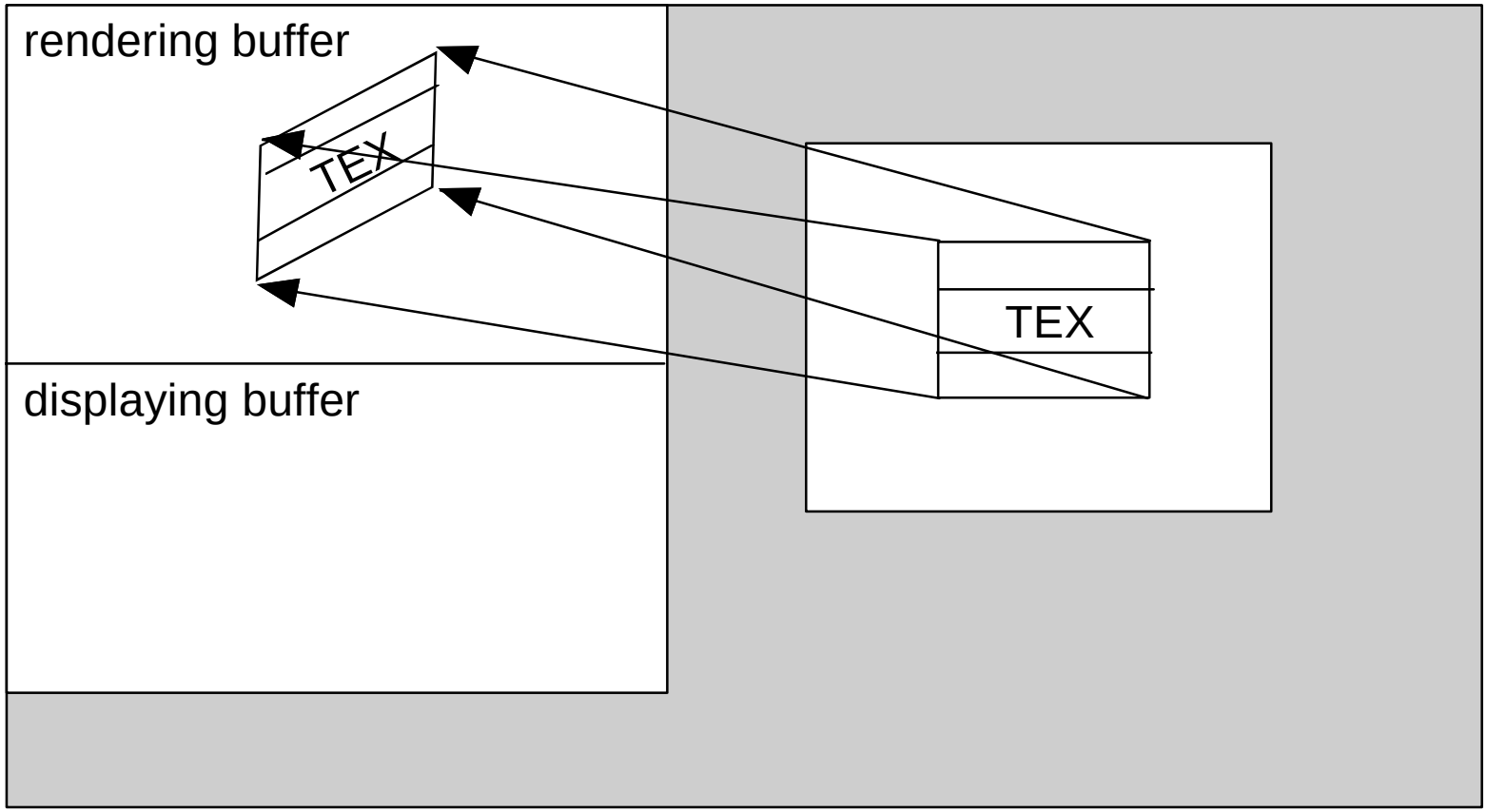


Primitive double buffer (3)



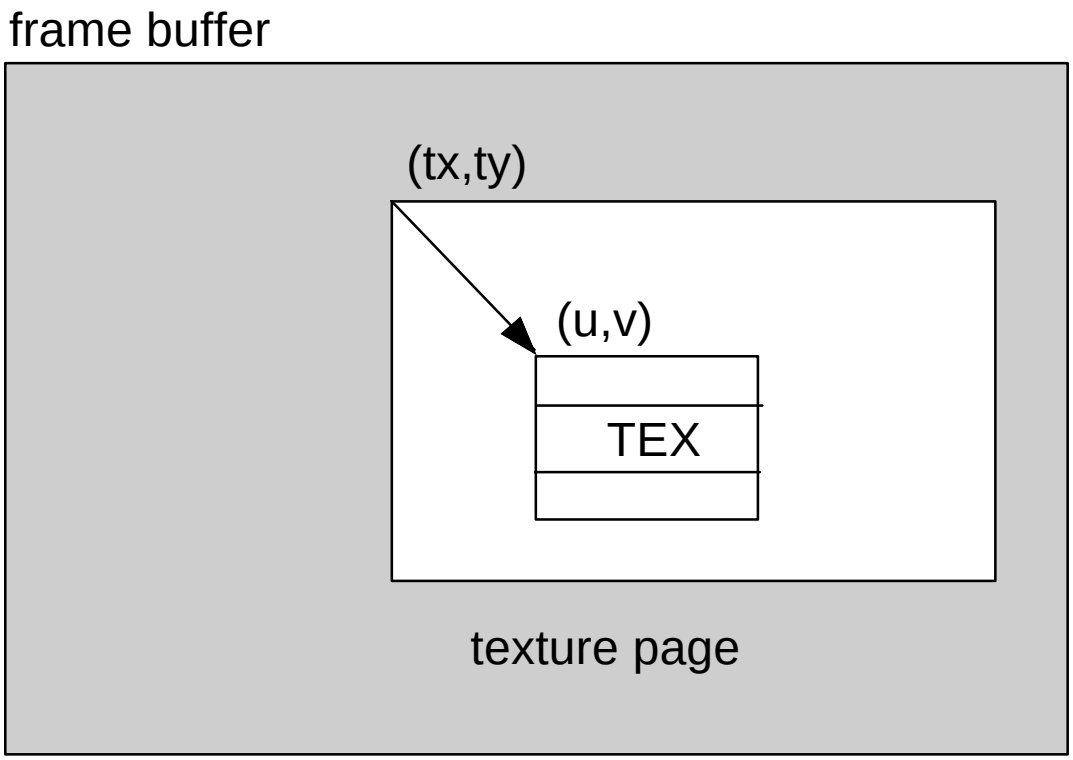
Texture mapping

A texture pattern in the frame buffer is pasted onto the surface of a polygon



Texture coordinates

Determined by texture page coordinates (tx,ty) and texture page offset (u,v)



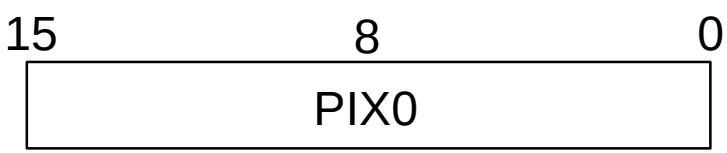
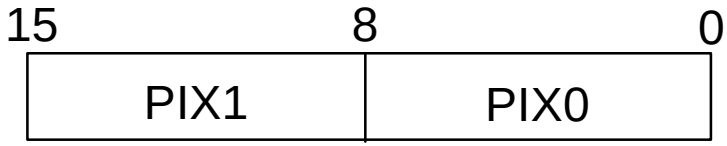
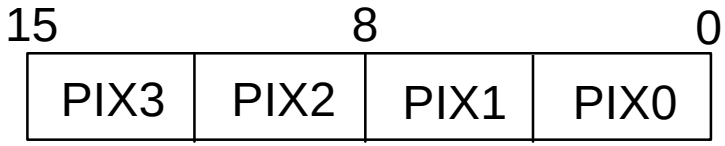
$$\begin{pmatrix} 0 \leqq u \leqq 255 \\ 0 \leqq v \leqq 255 \end{pmatrix}$$

Texture specification

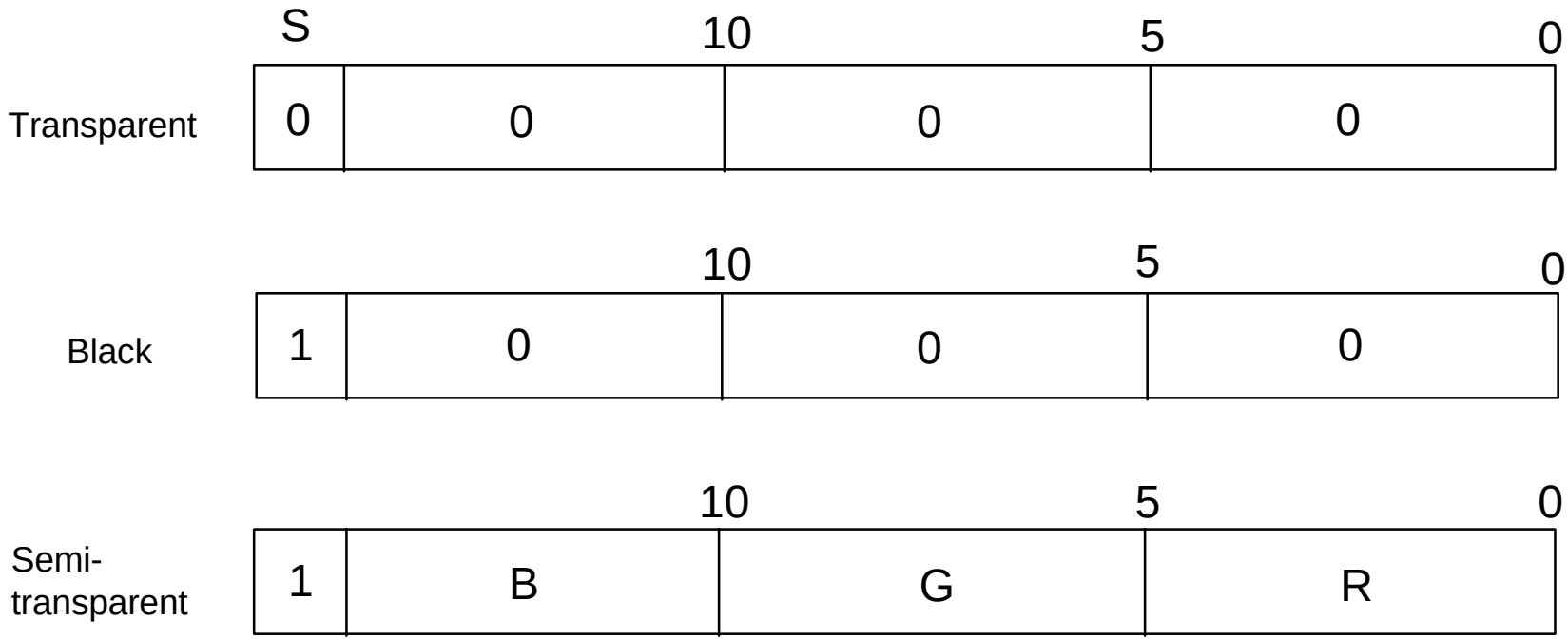
TAG			
CODE	B0	G0	R0
Y0		X0	
tpage		v0	u0
Y1		X1	
clut		v1	u1
Y2		X2	
		v2	u2

```
typedef struct {
    u_long tag;
    u_char r0, g0, b0, code;
    short  x0,     y0;
    u_char u0, v0; u_short tpage;
    short  x1,     y1;
    u_char u1, v1; u_short clut;
    short  x2,     y2;
    u_char u2, v2; u_short pad;
}  POLY_FT3;
```

Texture pattern format

mode	type	number of colors per polygon	pixel format
16bit	direct	32767	
8bit	CLUT	256	
4bit	CLUT	16	

Transparency control



Semi-transparency rate is specified by tpage

Rendering speed

Three basic rules

Fast ← Rendering speed → Slow

(1)	4bit	>	8bit	>	16bit
(2)	flat	>	sprite	>	texture polygon
(3)	non-transparent		>		semi-transparent

Blocking functions and non-blocking functions

- blocking functions

  → functions that wait for processing to finish and then return

- non-blocking functions

  → functions that return without waiting for processing to finish

    - DrawOTag() rendering
    - LoadImage() frame buffer access, etc.

  → operates while sharing CPU and memory

Program paradigm

```
while(1){
    /* swap buffer */
    i = ( i == 0 )? 1:0;
    /* set primitive buffer i */
    /* wait for rendering to finish */
    DrawSync(0);
    /* wait for VSync */
    VSync(0);
    /* begin rendering */
    DrawOTag ( buff[i].ot );
}
```

2D Graphics Sample

2D graphics sample demonstration

- 2D rendering with the PlayStation
- Creating data using the sprite editor
- Explanation of the sample source code using libgpu

2D drawing on the PlayStation

on **PS**

There is no hardware capability for displaying sprites

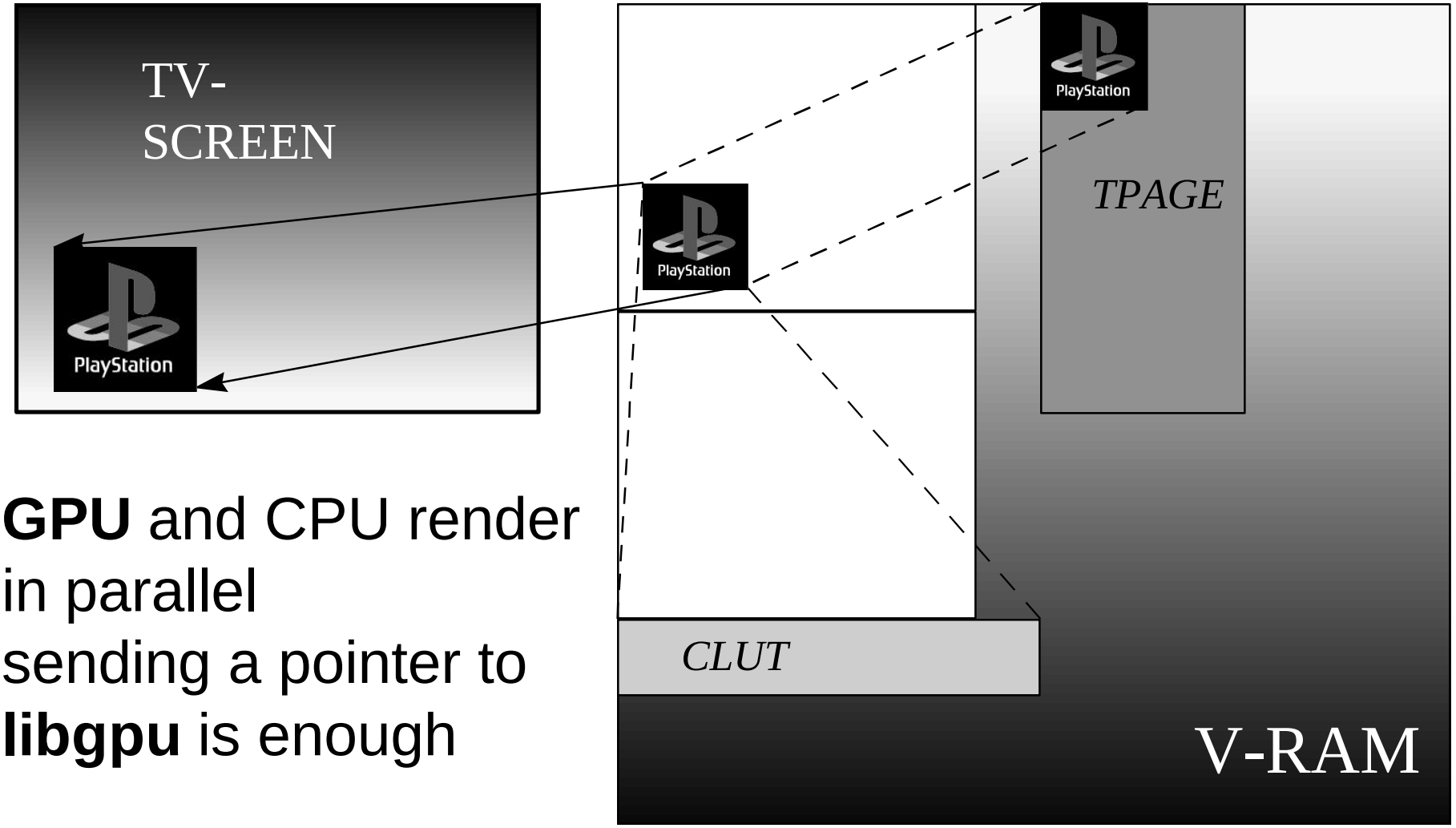
So

Express sprites as textured polygons

Then

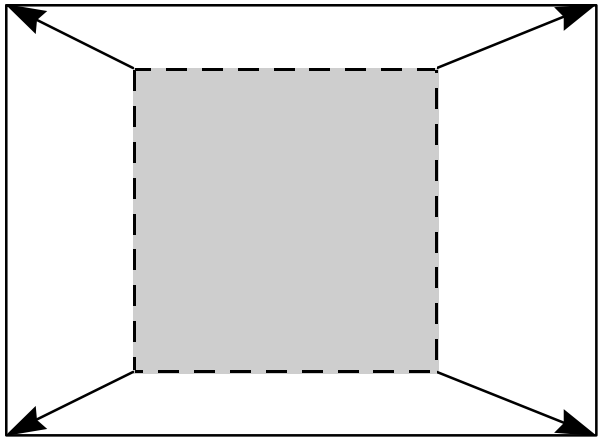
- Inversion, transparency, semi-transparency, palette, enlargement/reduction, rotation and general sprite expression are possible
- There are almost no size or shape restrictions! (The division of large characters into smaller parts)
- There is no horizontal direction limit! (There is no flickering and multiple joints are OK)
- Since it can hold a fixed image, a frame double buffer can be used! (It's OK to think of it in the same way as high-speed 'Put" routine that can perform parallel processing)

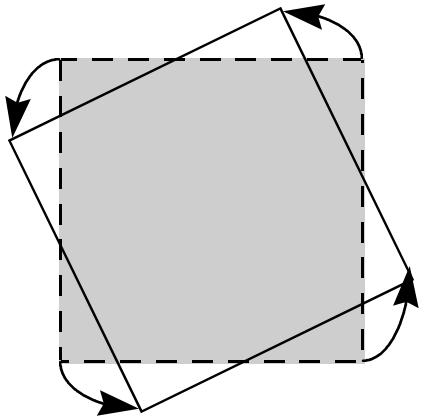
Displaying sprites



GPU and CPU render in parallel sending a pointer to **libgpu** is enough

Reduction, enlargement, and rotation

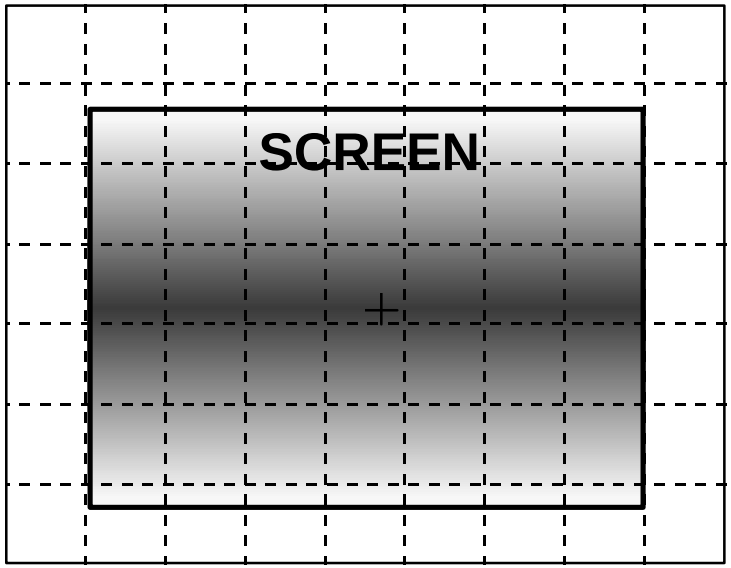




Set coordinates for four points

Complex rotations (rotations around X- or Y-axis, for example) are simplified by using **libgte**, which performs 3D matrix operations, etc.

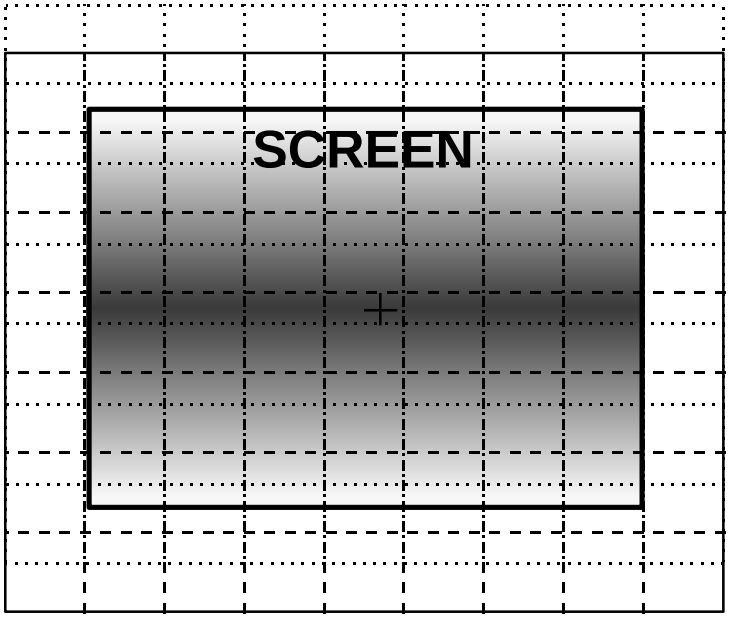
Using BG (backgrounds)



Polygons are packed in a mesh pattern

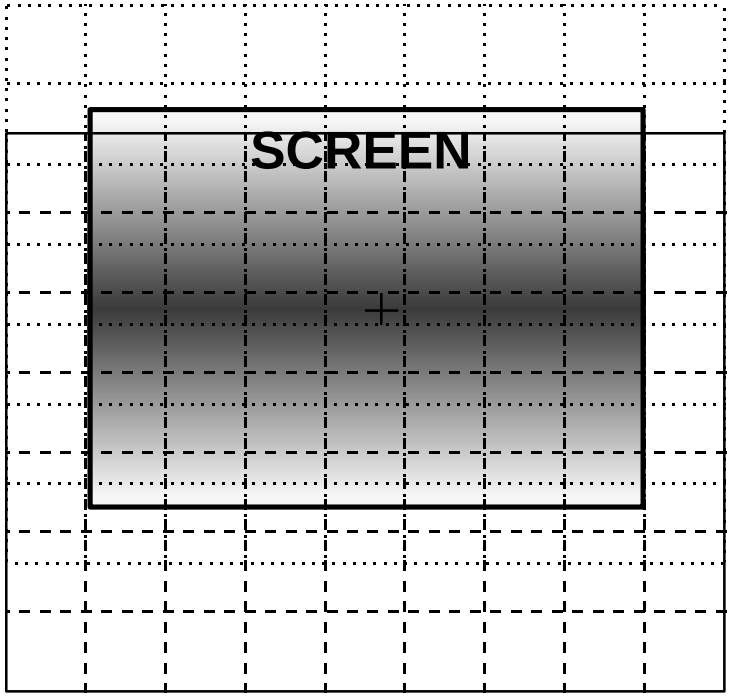
Each BG cel is pasted to a polygon as a texture

Using BG (scrolling)



Scrolling is performed by moving all the polygons in the scrolling direction

Using BG (Cel motion)

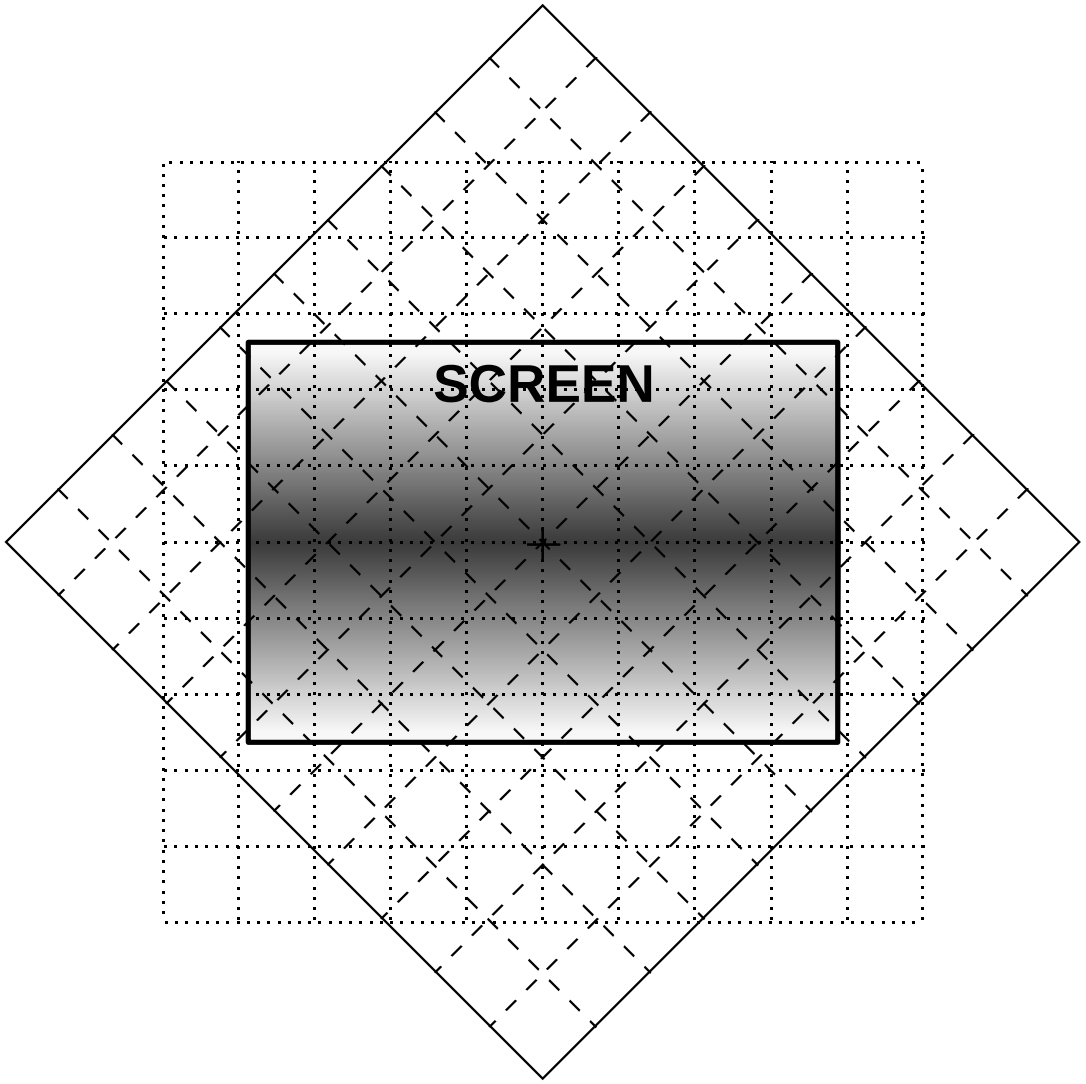


If the polygons keep moving, they will eventually be forced out since the polygons are packed in together

Before the polygons are forced out, the polygon textures are redefined and the polygon positions are adjusted

There must be at least one extra line's worth of polygons in the scrolling direction

Using BG (rotation)



When performing rotations, the screen must be filled no matter what
rotation angle is used

This must also be taken into consideration for scrolling BG

Creating data using the sprite editor

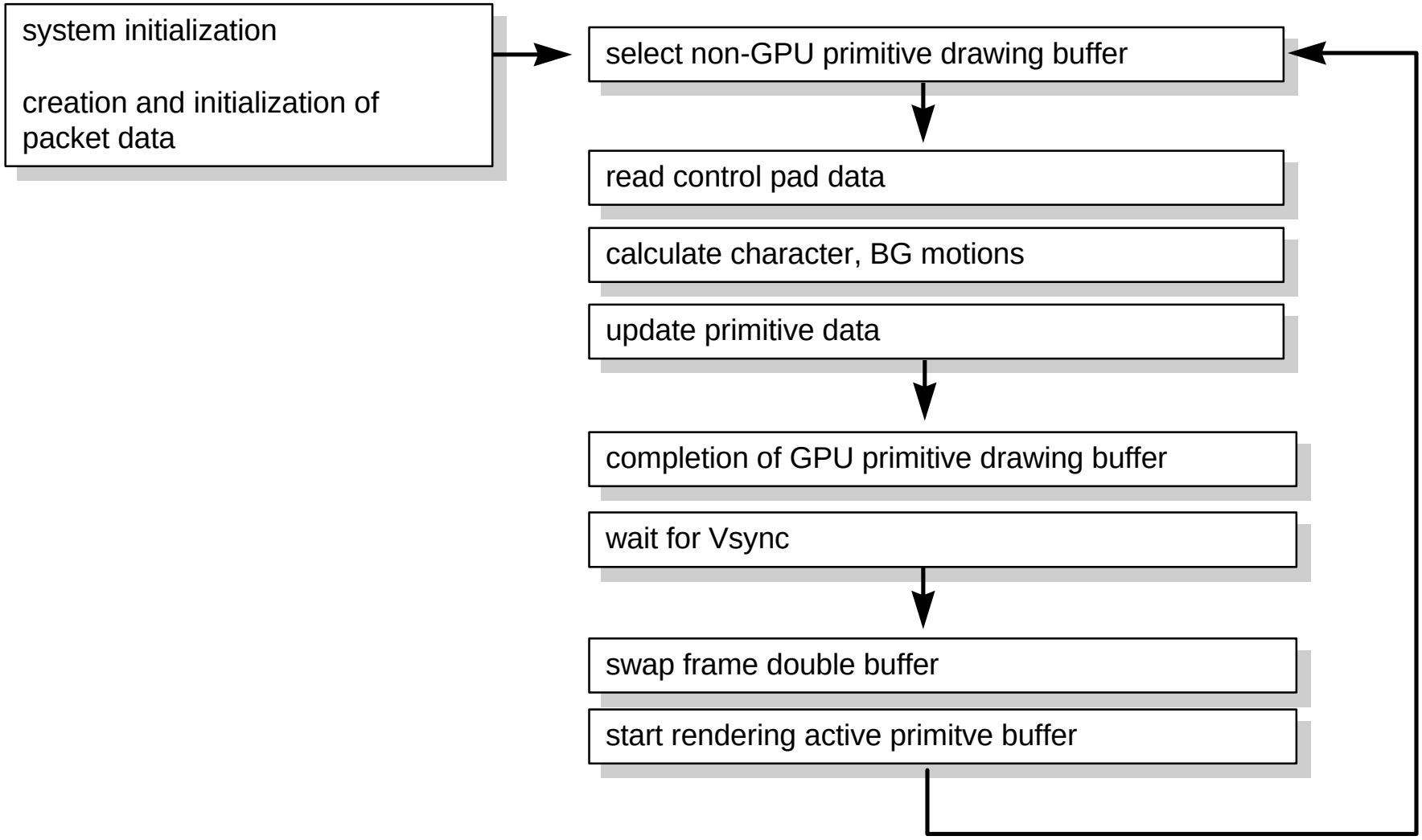
.***TIM***: pixel image, palette data

.***CEL***: image position data for map display

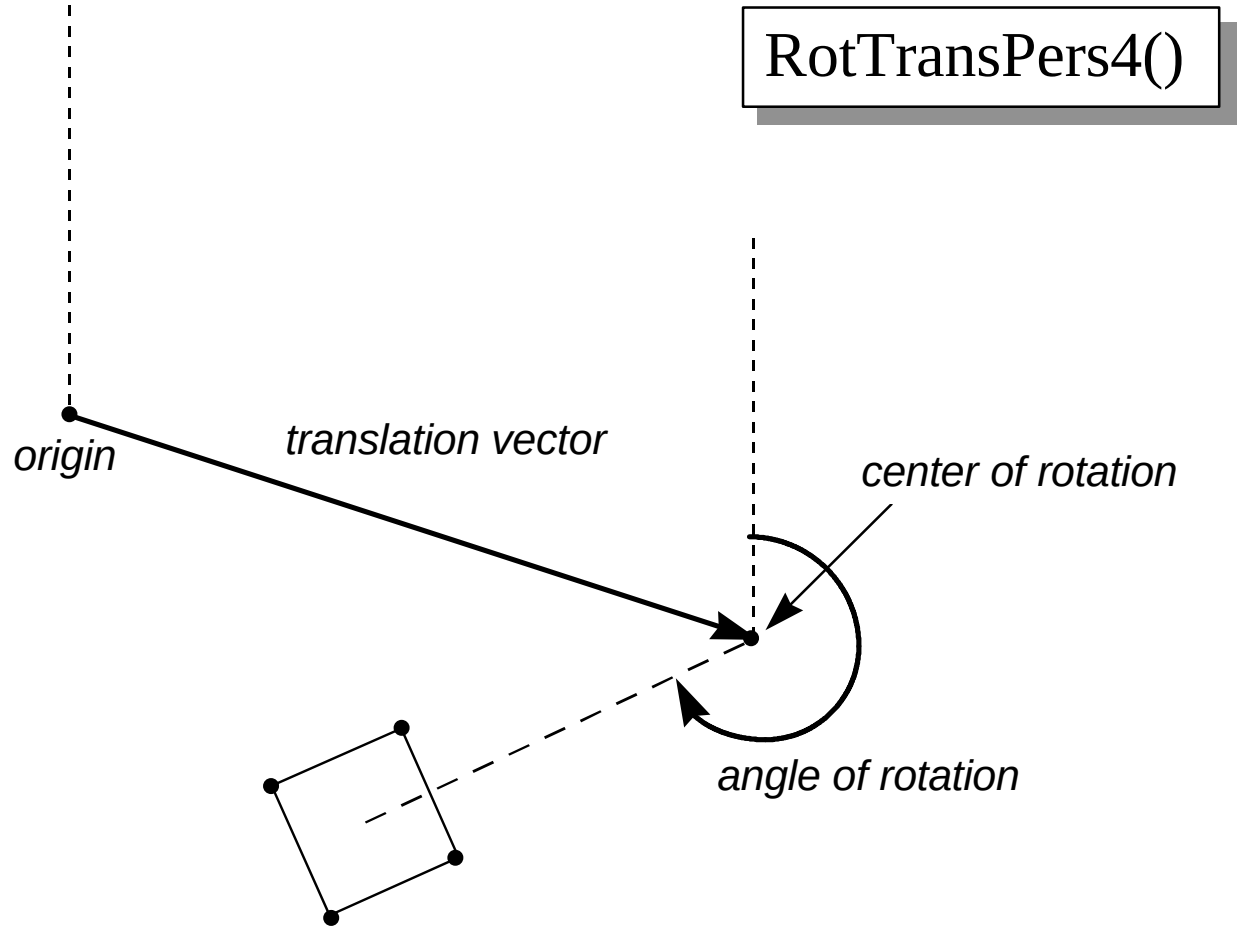
.***BGD***: cel position data for map display

.***ANM***: image position data and tie sequences for sprite animation

Explanation of the sample (outline)

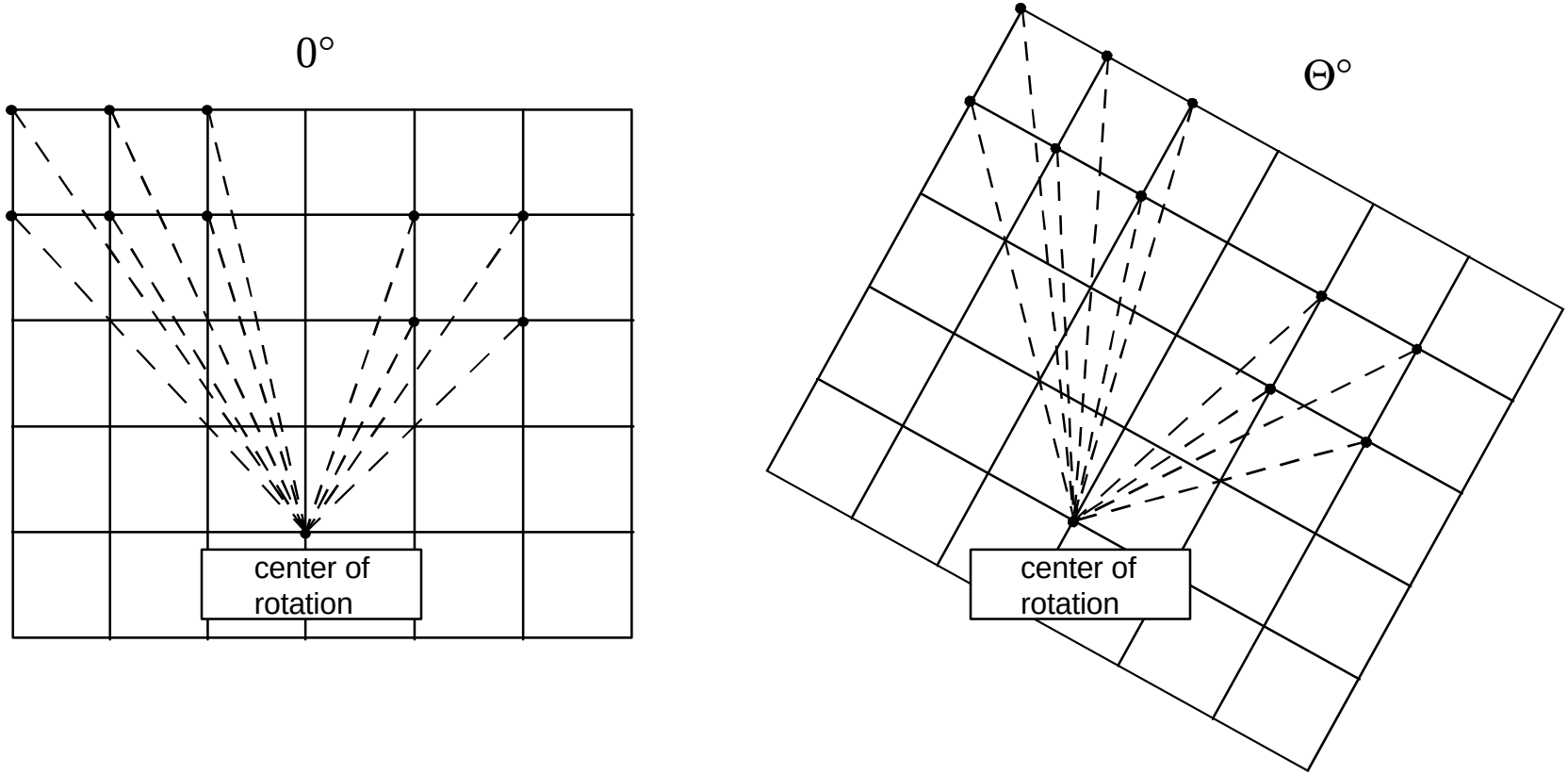


GTE coordinate transformations



The coordinates for these four points can be determined all at once

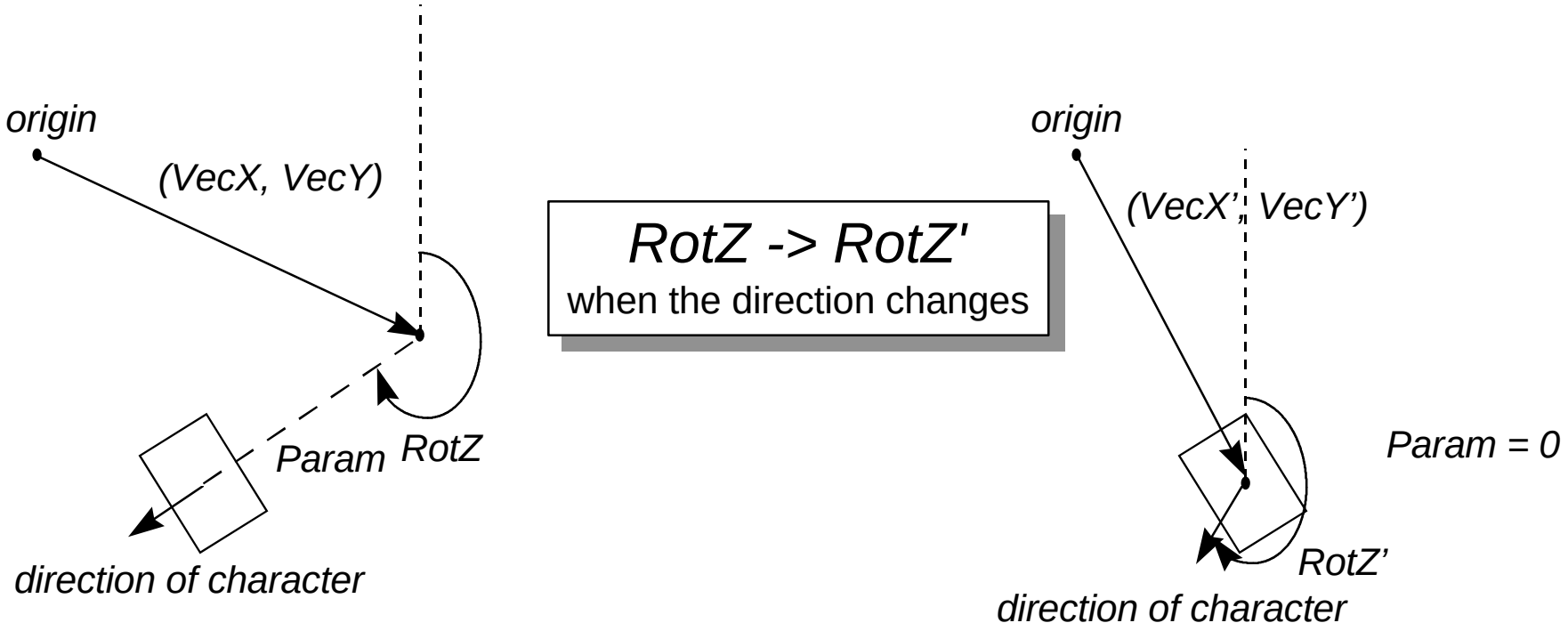
Rotation of BG



With translation vector (x,y)=(0,0), each of the polygons are rotated using the same rotation matrix and the offsets from the center of rotation...this rotates the entire BG.

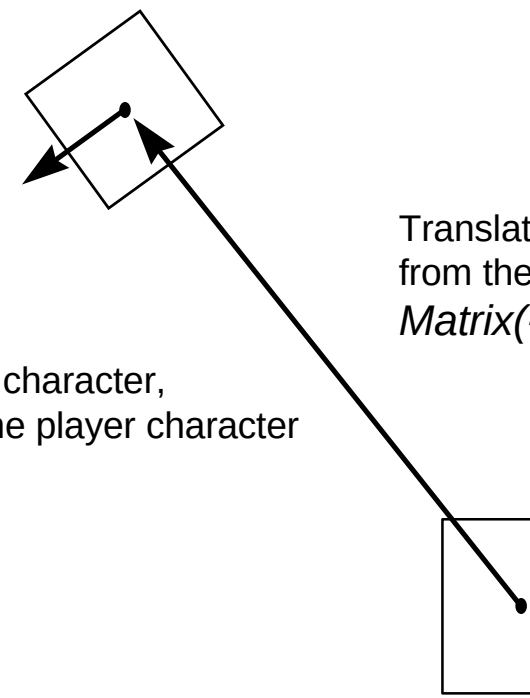
Moving enemy characters

- character should be able to move fully in 360 degrees
- coordinate transforms should be made easy with BG rotation



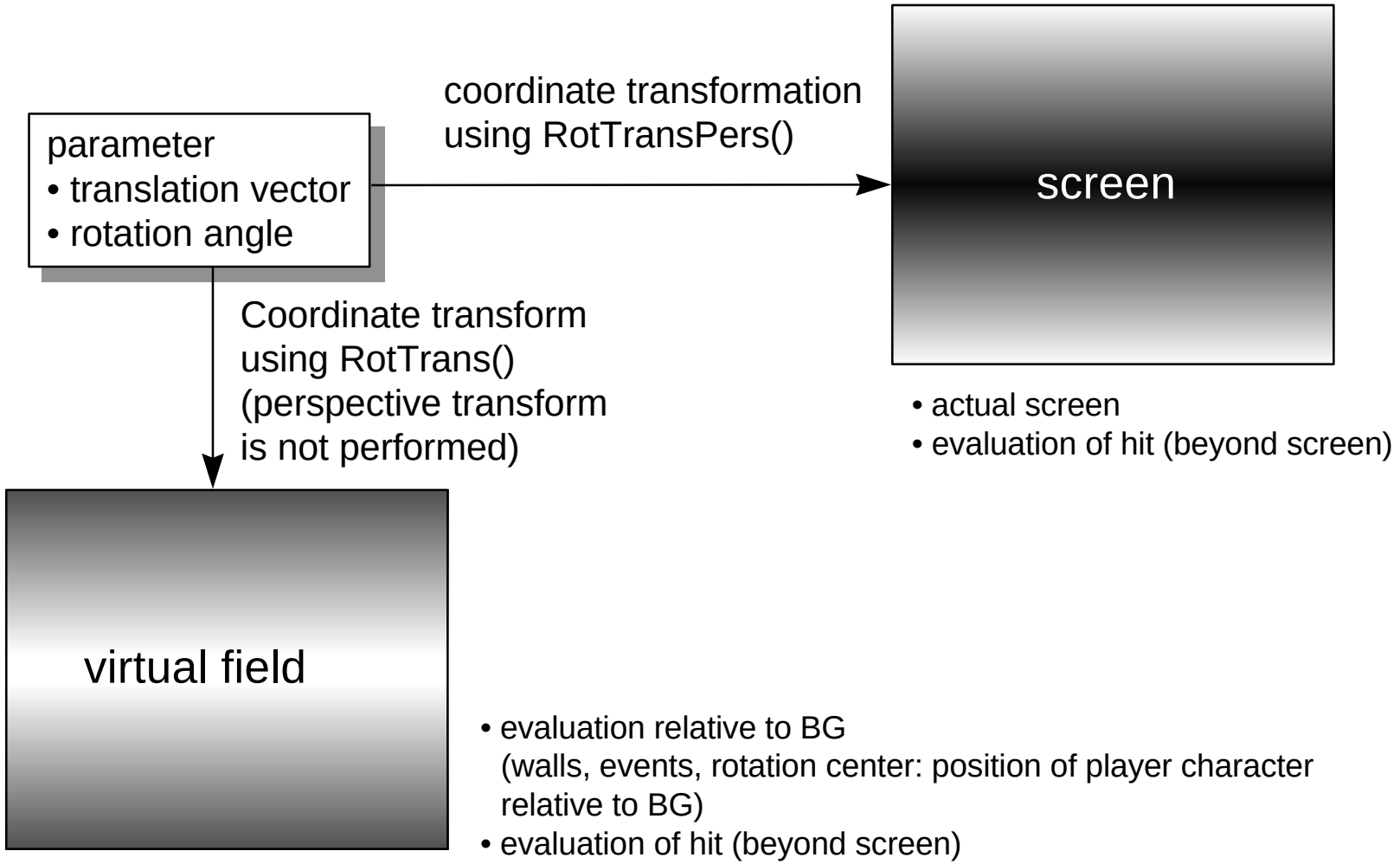
(an overflow may result depending on the size of the Param)

Coordinate transforms based on the player character



A coordinate transform (RotTransPers()) should be performed using this translation vector and rotation angle (Actually, the offset for the **Param** must be included for the enemy character)

Coordinate transforms for different screens



Introduction to 3D Graphics

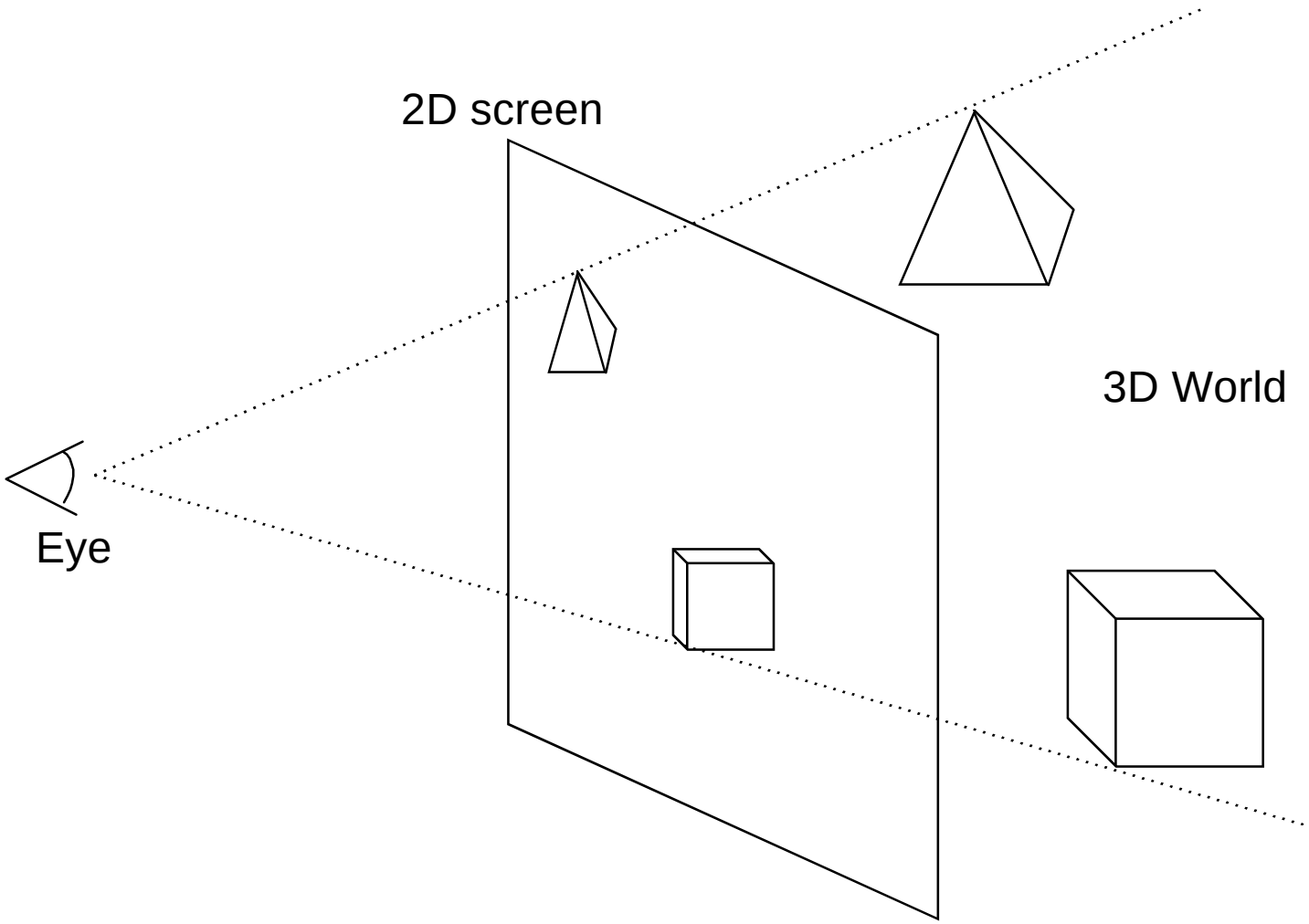
2D vs. 3D graphics

- 2D graphics
  - use sprites and backgrounds
  - all images are prepared beforehand
  - almost no calculations are needed for generating images

2D vs. 3D graphics (cont)

- 3D graphics
  - use mathematical models
  - images are generated through calculations
  - calculations are needed for generating images

3D display



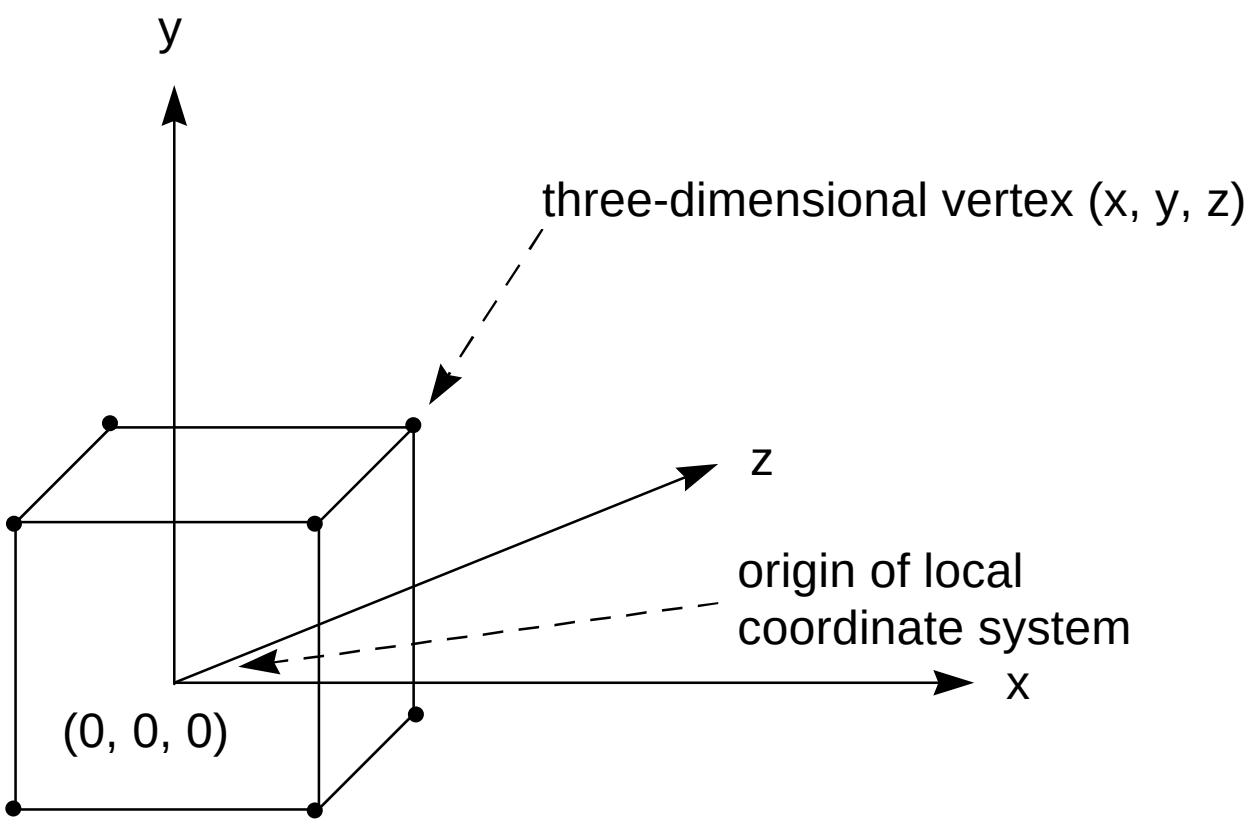
Generating 2D images from 3D data

- Defining 3D objects
- Arranging objects in 3D space
- Defining view point
- Projection to a 2D screen
- Rendering 2D images

Defining 3D objects

- objects are defined in three-dimensional vertex coordinates
- surfaces are defined through multiple vertices
- definitions are performed with local coordinates

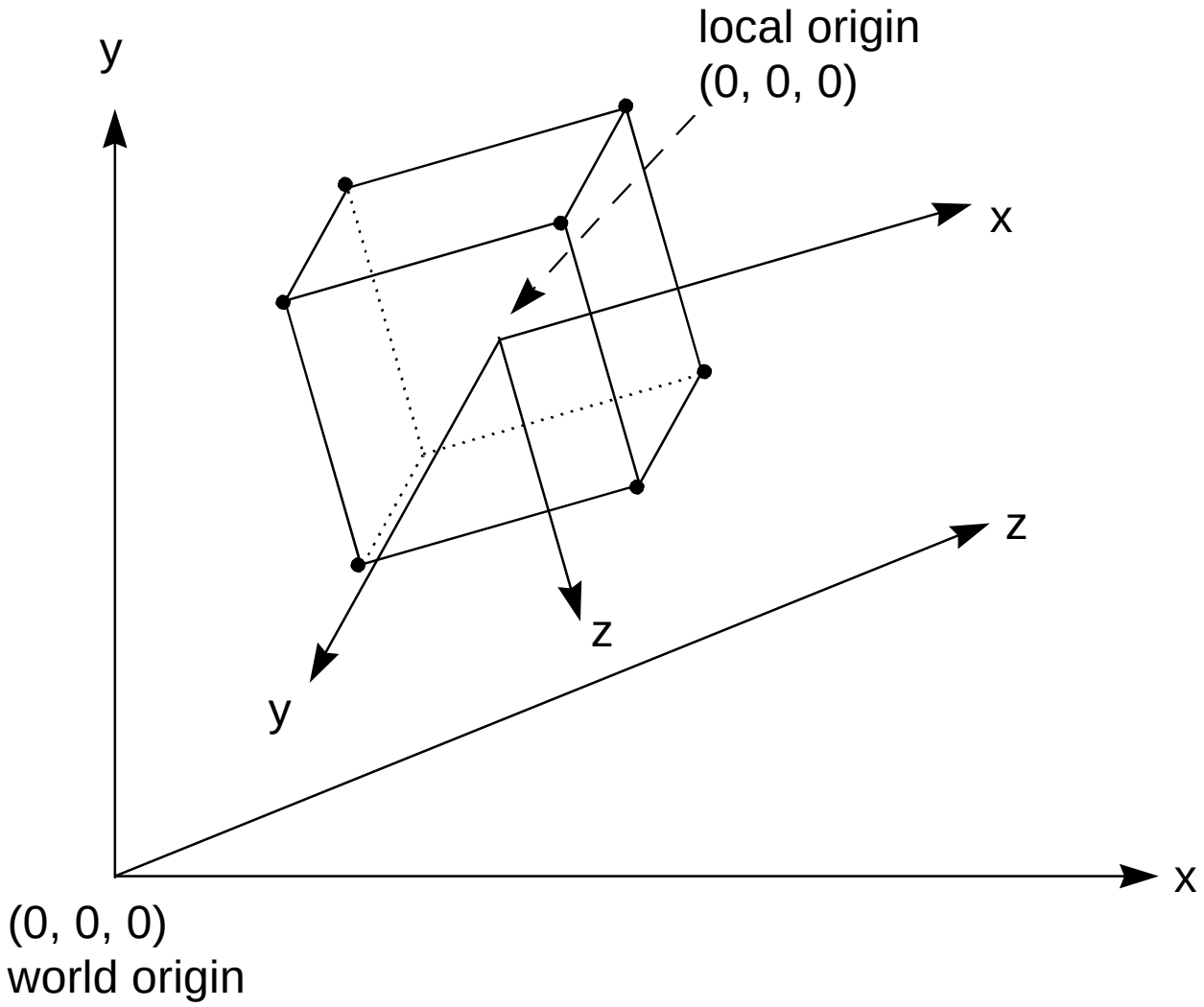
Defining 3D objects (cont)



Arrangement of objects in 3D space

- positioning using translations
- orienting using rotation
- 3D space is defined with the world coordinate system

Arrangement of objects in 3D space (cont)



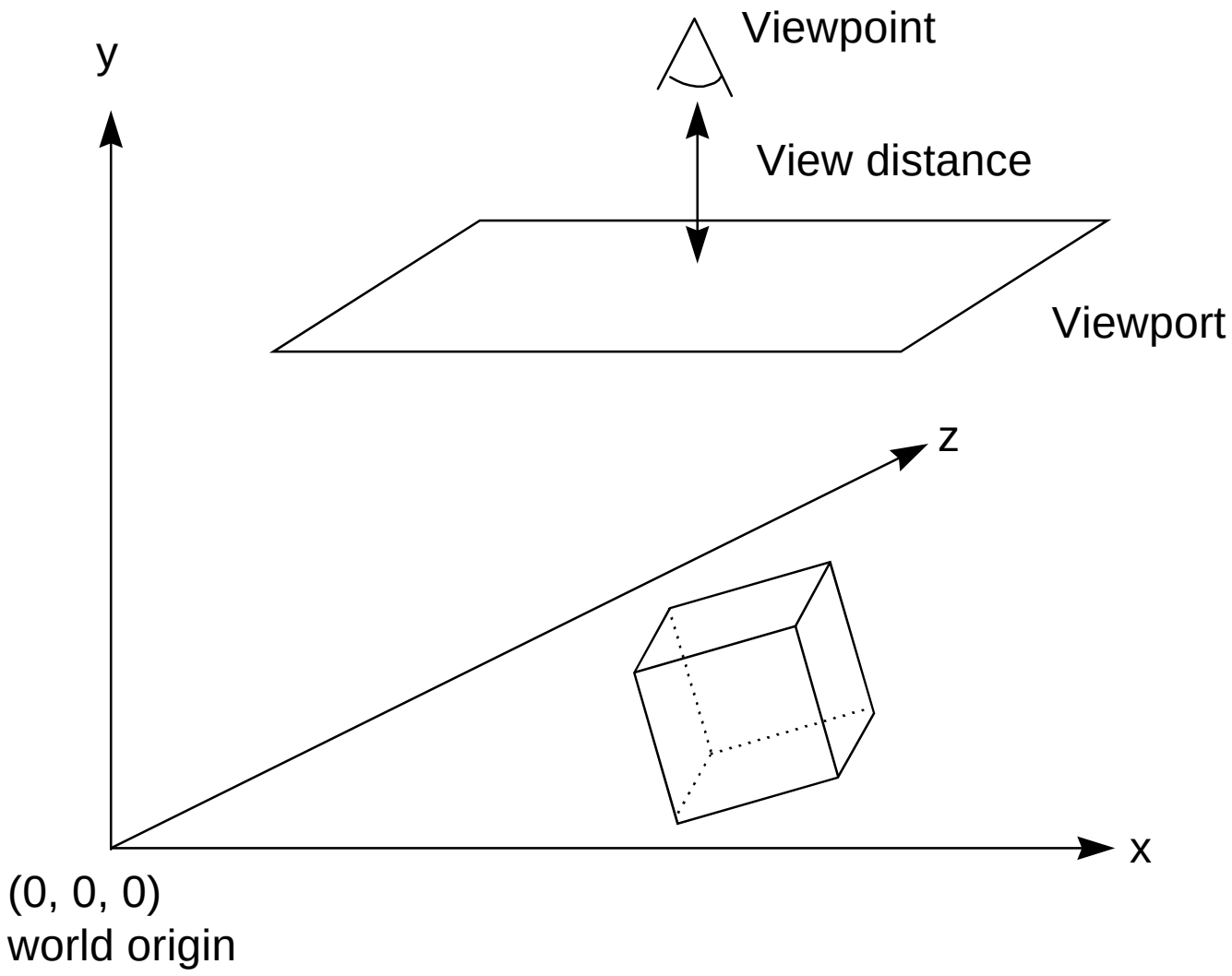
Matrix calculations

- translations, rotations are defined with matrices
- all vertices are multiplied with the following matrices:
  - local-to-world matrix
  - world-to-screen matrix

Defining the view point

- the view point is defined in the world coordinate system
- the view point is defined with the translation and rotation of the world/screen matrix
- the viewport is defined according to the distance to the screen

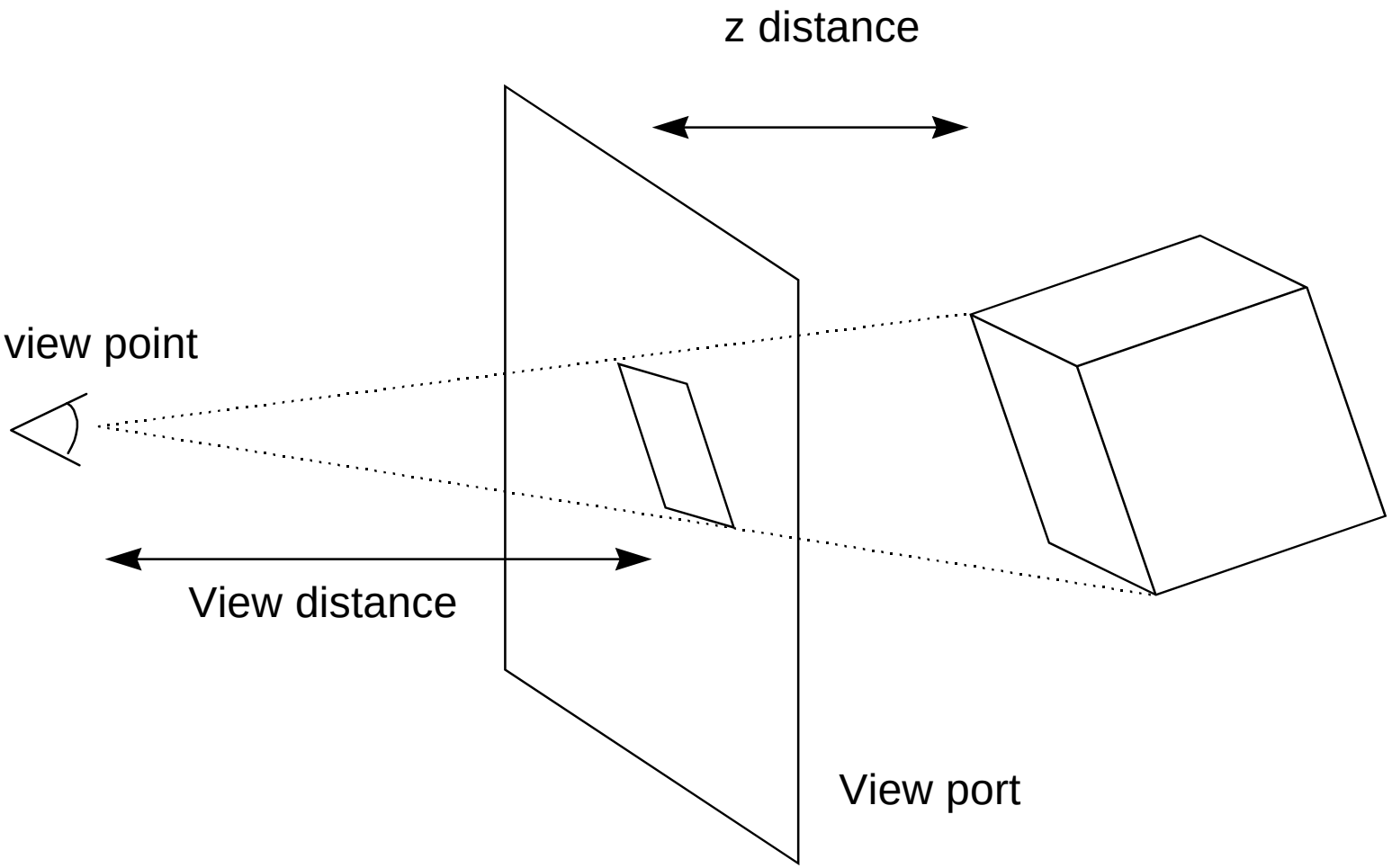
Defining the view point (cont)



Projecting the world space to the screen

- projection to the screen is performed after rotation and translation has been performed on all the objects
- vertices are projected with perspective transformations
- this results in the final screen coordinates

Projecting the world space to the screen (cont)

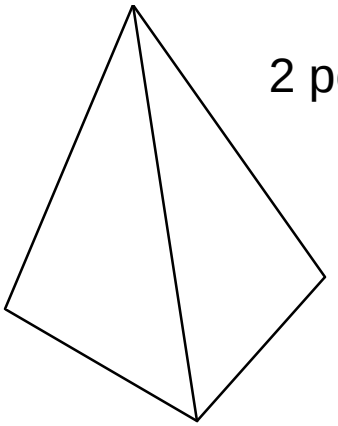


Rendering 2D images

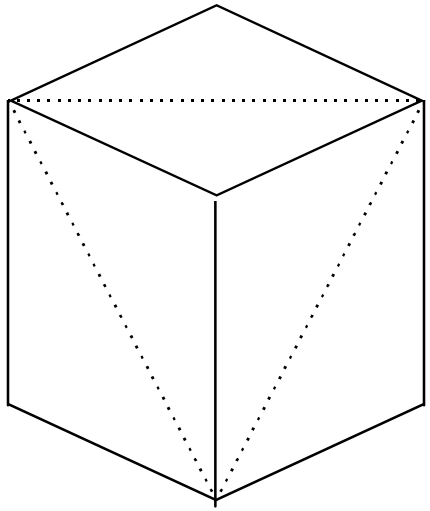
- 2-dimensional points making up the surface of an object are determined
- these surfaces are called polygons
- these polygons are generally triangles

Rendering 2D images

2D polygon



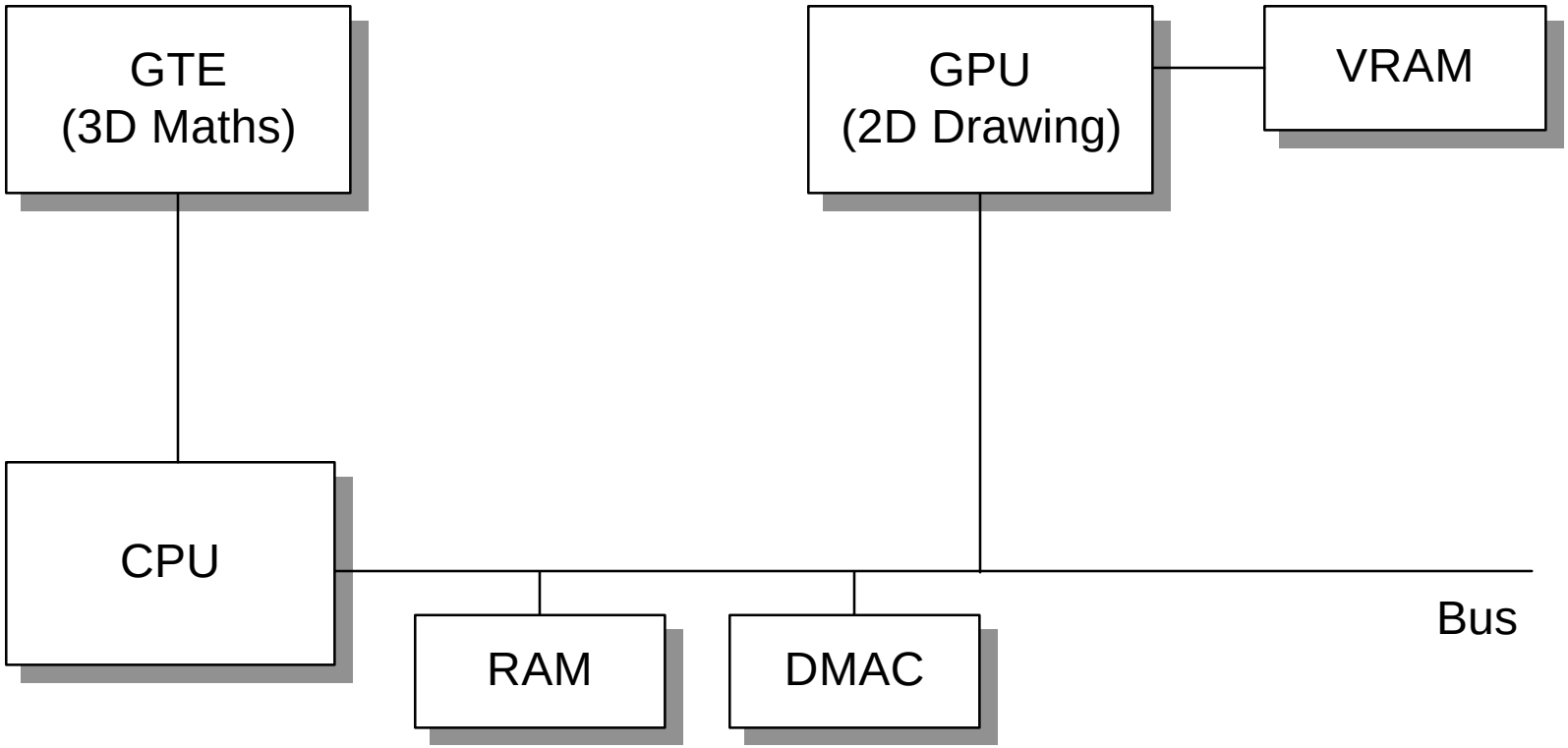
2 polygon



6 polygon

An introduction to GTE

PlayStation hardware configuration



Geometry transfer engine

- custom coprocessor for performing the following calculations:

  - 3D -> 2D coordinate transforms
  - light source calculations
  - depth cueing
  - 3D animation

GTE library

- functions to access the GTE hardware
- eliminates the need for programming 3D calculations
- much faster than using the CPU

3-dimensional coordinate transform functions

- executed after initialization
- performs rotation, translation and perspective conversion of coordinates (RotTransPers)
- 3 or 4 vertices calculated simultaneously

3-dimensional coordinate transform functions (cont)

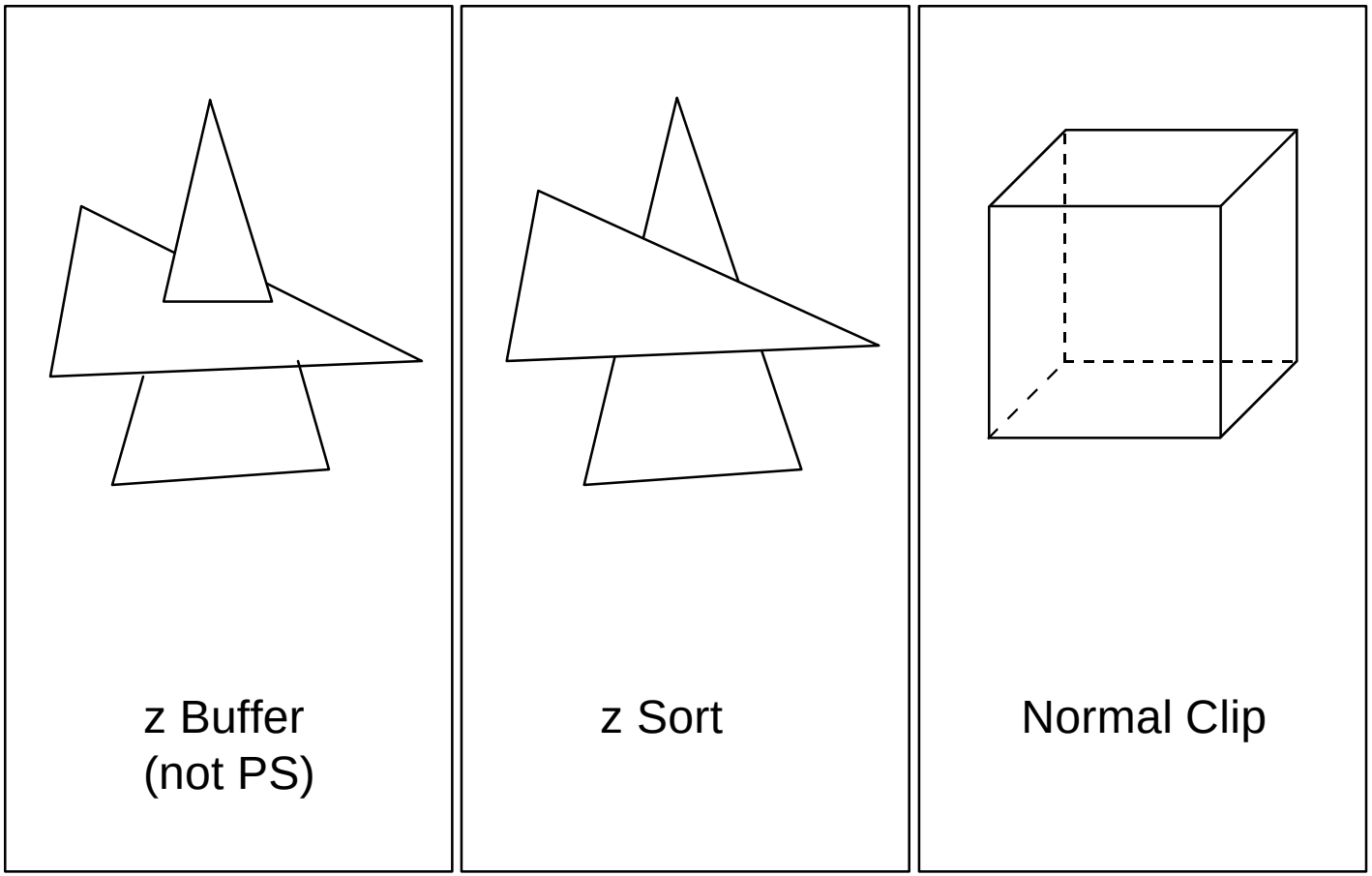
- other functions
  - Z-average function
  - ray clipping
  - light source calculations
  - depth cueing

Example of 3D coordinate transformation

```
SVECTOR     v0, v1, v2;          /* input 3D */
long        sxy0, sxy1, sxy2;    /* output 2D */
long        p;                   /* output depth data */
long        flag;
[some code here...]

RotTransPers3(&v0, &v1, &v2, &sxy0, &sxy1, &sxy2, &p, &flag);
[some code here...]
```

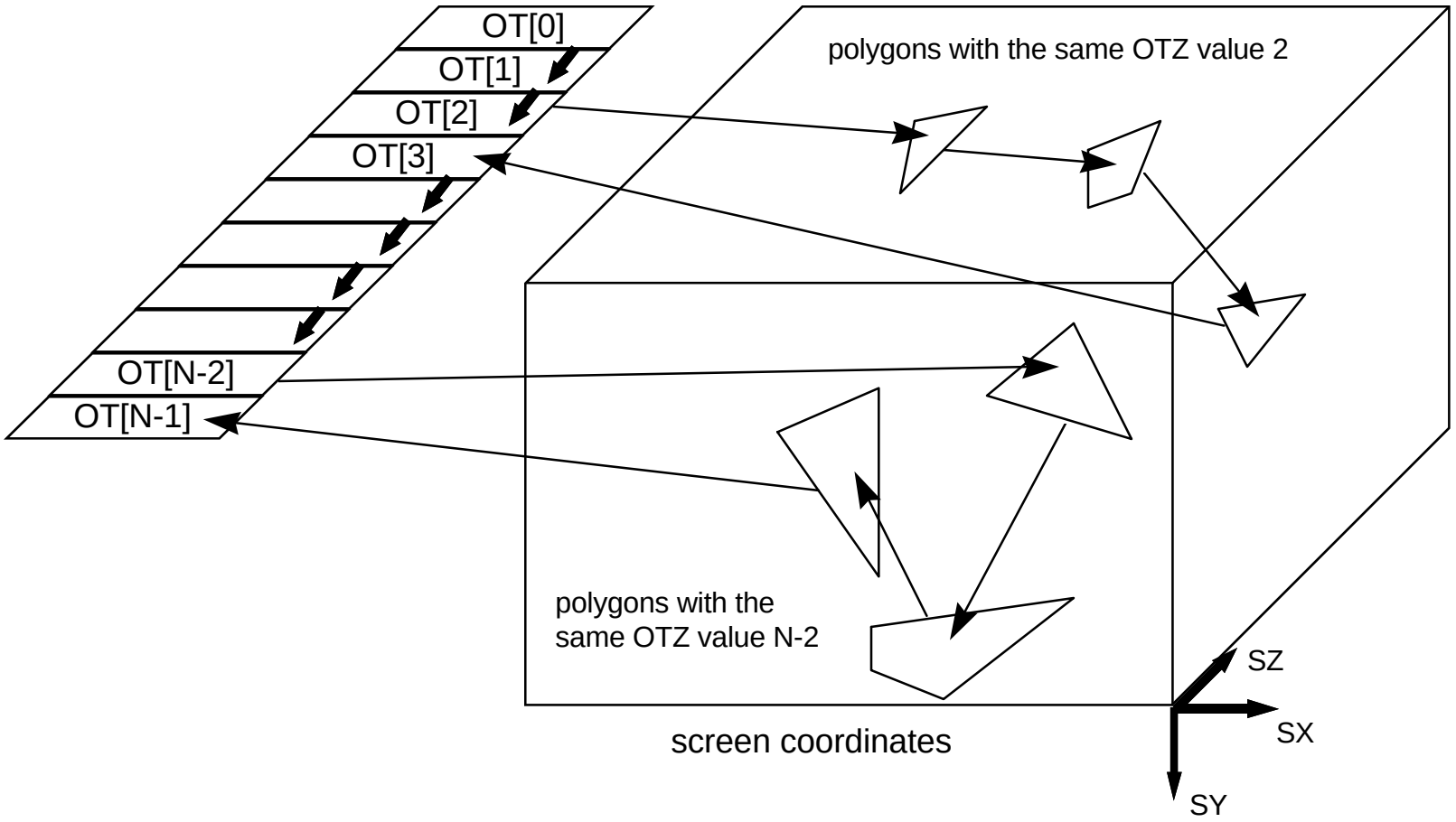
Hidden surface handling



Z sort

- rendering is performed beginning with the further polygons
- polygon is sorted by Z values
- the average Z value of the vertices is used
- the ordering table (OT) is used

Z sorting

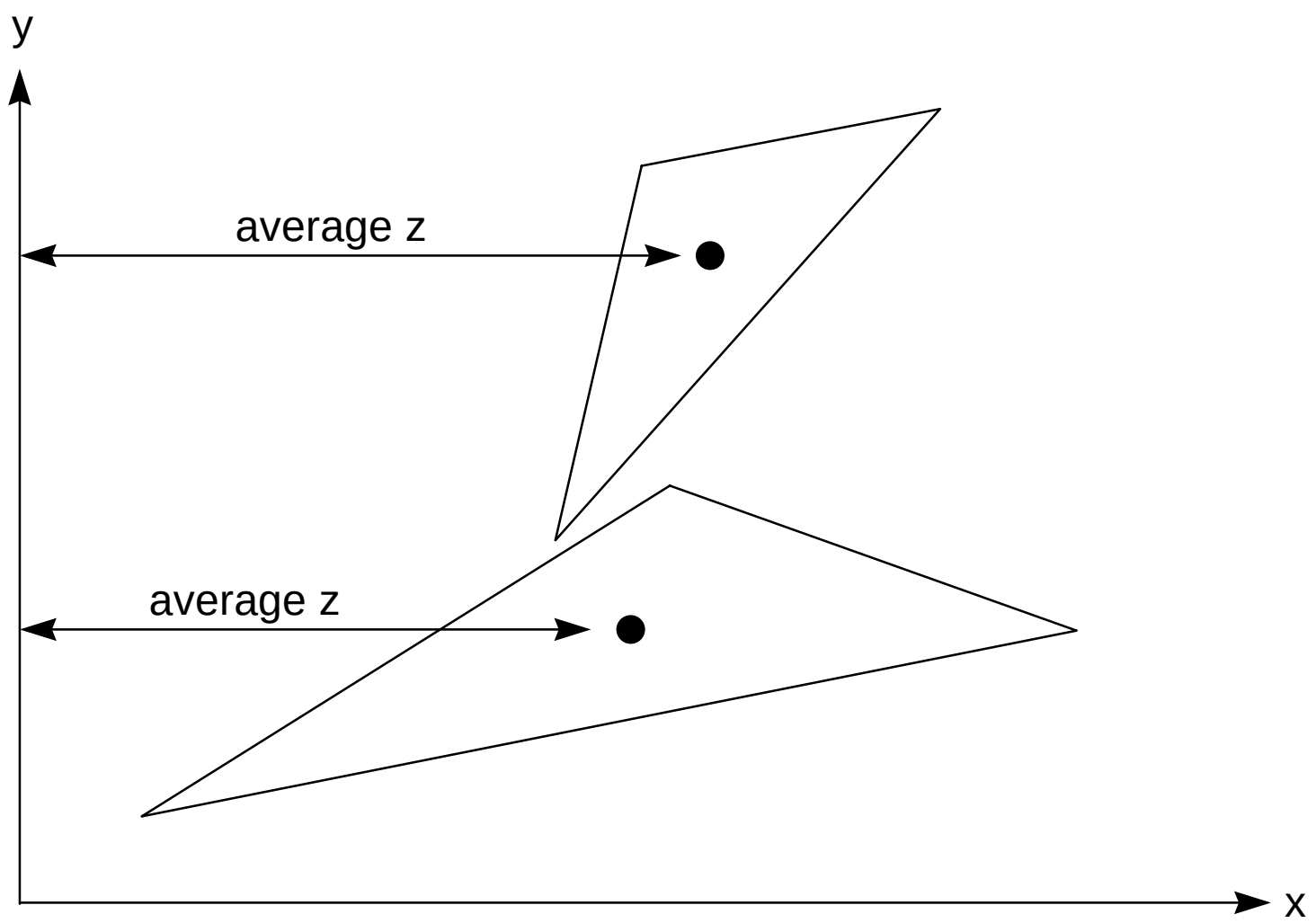


- the SZ value divided by a certain constant is called the OTZ value
- the OTZ value becomes an OT entry
- increasing the constant used to divide the SZ value makes sorting less precise

Z sort problems

- polygons cannot intersect
- mistakes in sorting can take place
- polygons with similar Z values can cause flickering

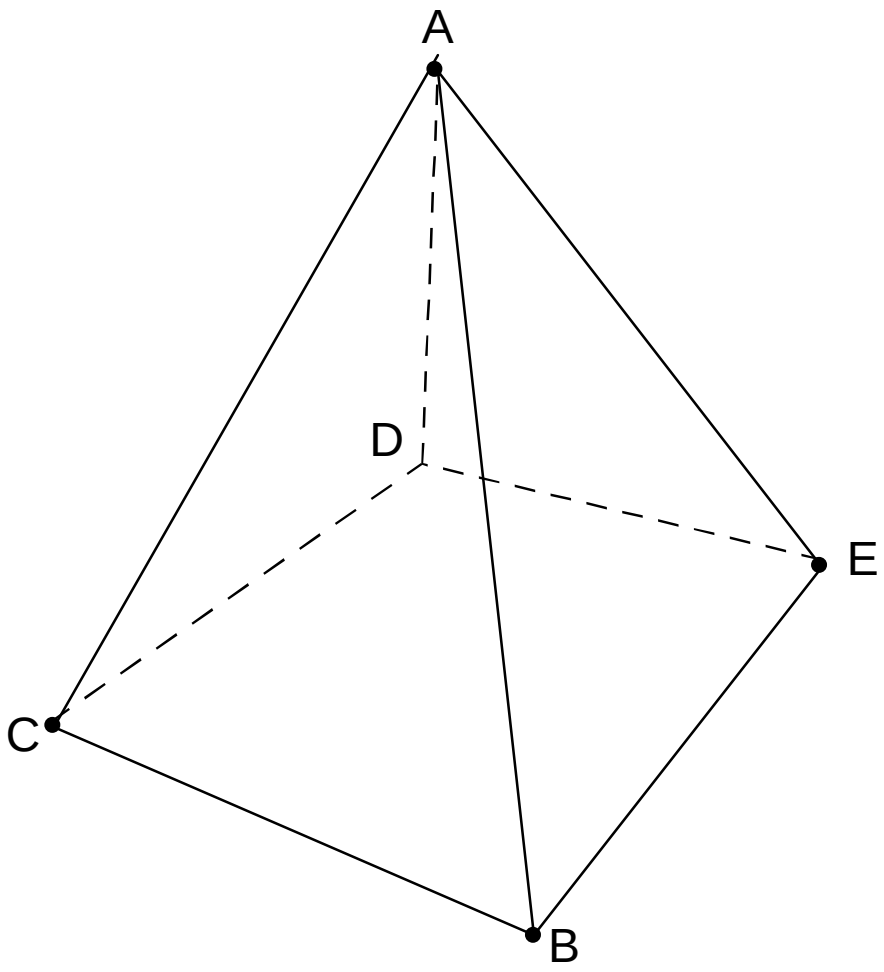
Z sort problems (cont)



Ray clipping

- eliminate surfaces oriented toward the rear
- vertices are ordered clockwise
- polygons that are clockwise on the screen are rendered
- polygons that are counterclockwise on the screen are clipped

Ray clipping (cont)



2D	3D
CW	A,B,C
ACW	A,C,D
ACW	A,D,E
CW	A,E,B

Clipping

- GTE functions return data for clipping in flag
- the GTE library contains software clipping functions
- the GPU performs 2-dimensional clipping

The GTE flag

- the GTE flag returns information such as

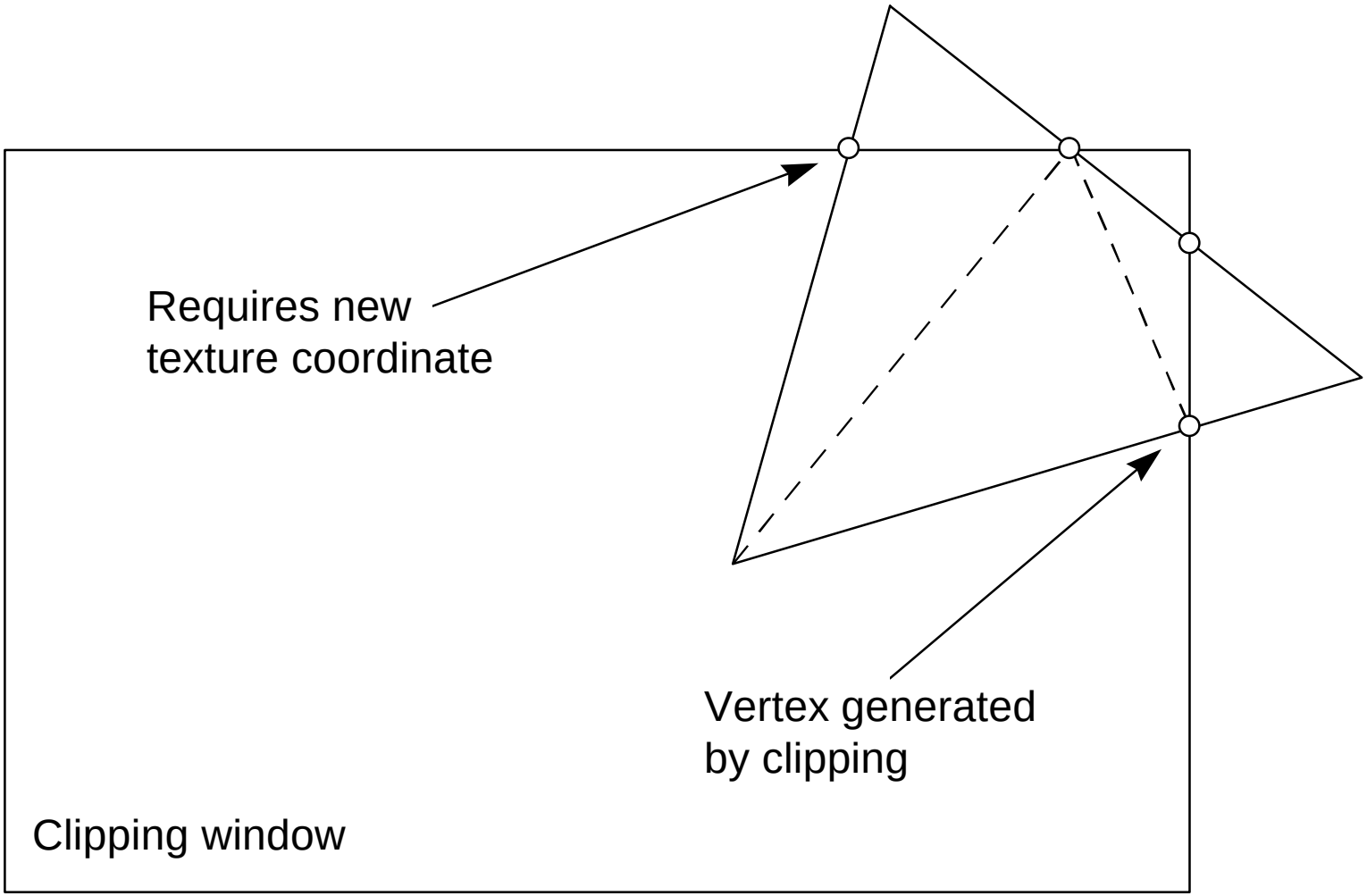
  - whether the boundaries of the screen coordinate system have been exceeded

  - whether the depth cueing area has been exceeded

3-dimensional clipping functions

- polygons are clipped by viewing volume
- one polygon is divided into multiple polygons
- in textured polygons, the textures are also divided

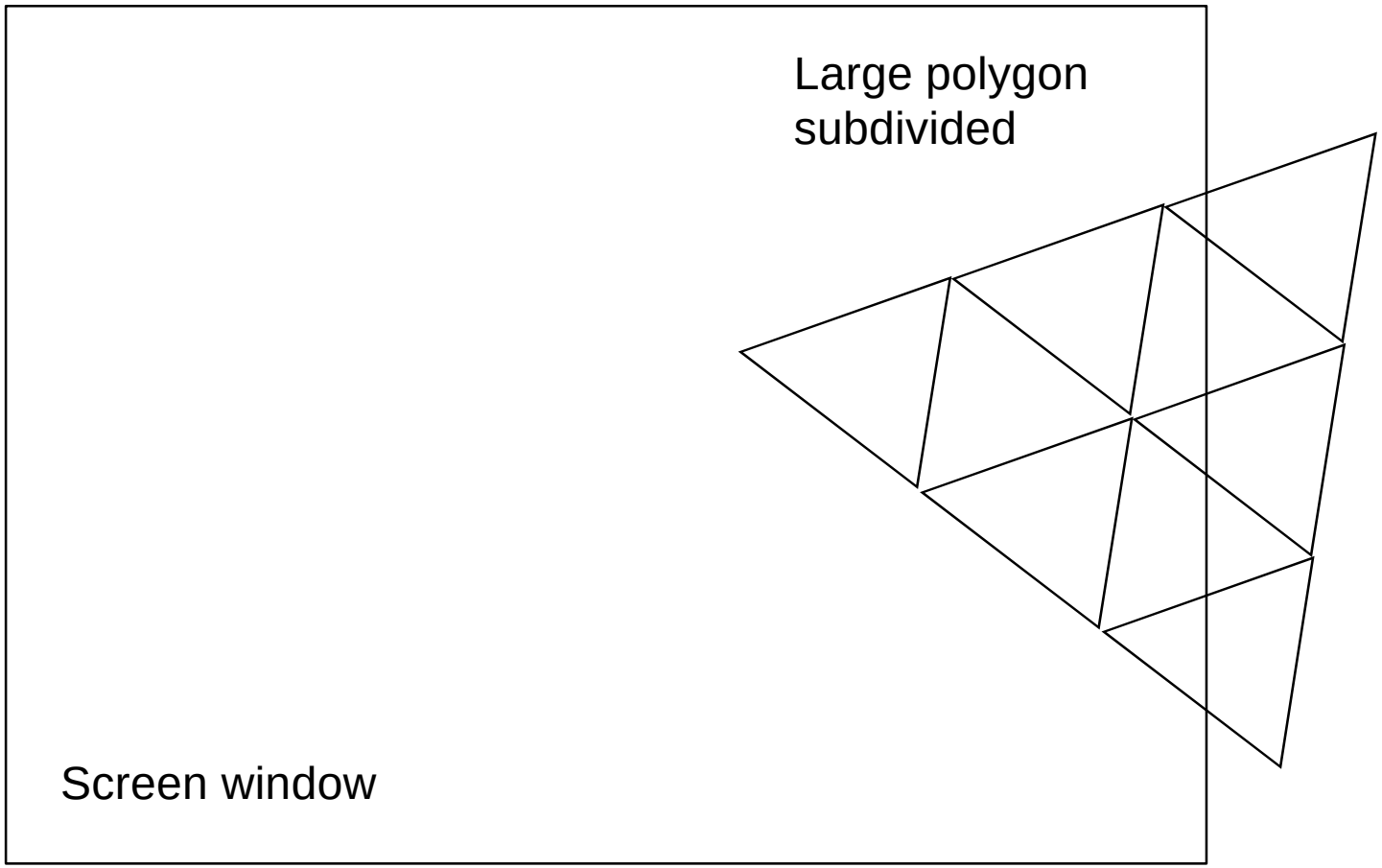
3-dimensional clipping functions (cont)



Polygon division

- acts in place of 3D clipping
- large polygons are divided into small polygons
- small polygons are clipped by the GPU
- warping in texture mapping can be decreased

Polygon division (cont)



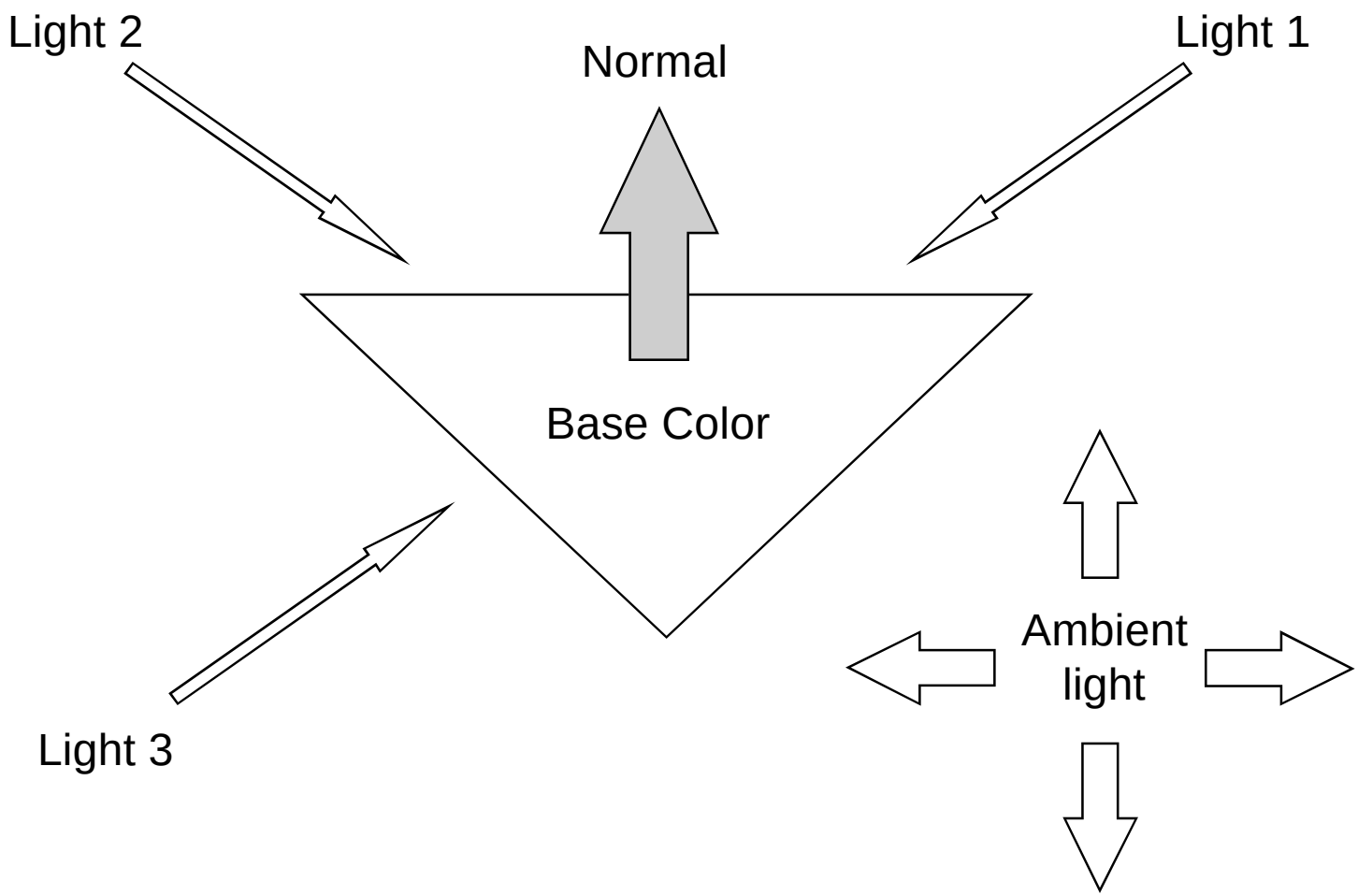
Light source calculations

- the GTE allows real-time light-source calculations
- 3 light sources + ambient light can be used at high speeds
- a light source is a flat light source from an infinite distance
  - spot lights are not supported

Light source calculations (cont)

- the polygon rendering color depends on the orientation of the polygon and the orientation of the light source
- the orientation of a polygon is defined by a ray
- ambient light is determined regardless of the orientation

Light source calculations (cont)



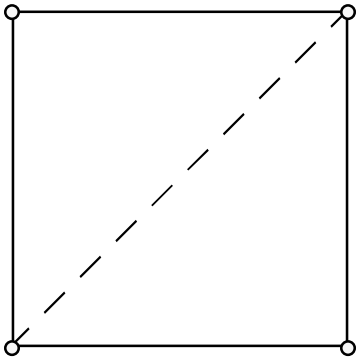
Depth cueing

- polygon colors are changed according to the Z value
- a fog effect is used
- the starting point and ending point of fog can be defined

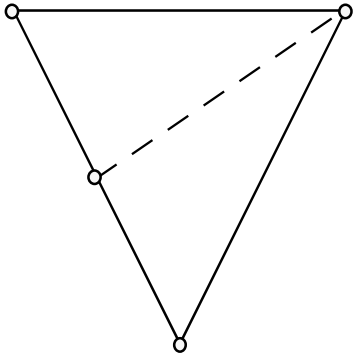
MIMe

- motion is defined by multiple key frames
- motion is displayed by interpolating between key frames
- deformation of model shapes is also possible (vertex MIMe)

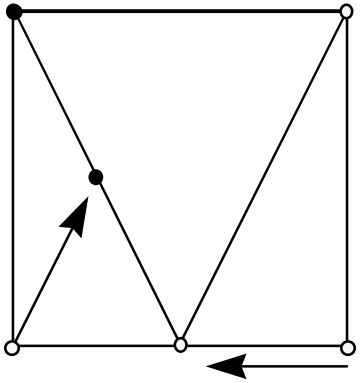
MIMe (cont)



Frame A



Frame B



Vertex paths

MIMe (cont)

- multiple interpolation of two or more key frames
- compose using key frames having minimal complex motions
- MIMe can be used in applications other than those involving vertices

Optimization of three-dimensional calculations

- optimization through functions that perform simultaneous operations on three vertices

RotTransPers3 < RotTransPersX3

Optimizing with common vertices

- method:

  coordinate transformation is performed one at a time on a vertex group

- advantage:

  screen coordinates can be determined quickly

Optimizing with common vertices (cont)

- disadvantages:
  - vertices are separated from the polygon
  - this makes it necessary to do calculations at the polygon level separately
  - this increases memory access

Optimizing with independent vertices

- method:
  - perform vertex operations at the polygon level

- advantages:
  - suited for GTE hardware
  - other calculations for a polygon unit can be performed at the same time as coordinate transformations

Optimizing with independent vertices (cont)

- disadvantages:
  - it is possible for a vertex to have two or more coordinate transformations performed upon it

Optimizing ray clipping

- improve GTE and GPU performance
- generation of screen coordinates for clipped polygons is skipped
- only three vertices are used for four-vertex functions

Optimizing light-source calculations

- light source calculations are time consuming

- therefore:
  - use only Gouraud shading
  - perform light-source calculations only on polygons to be rendered
  - perform light-source calculations only during modeling

Optimizing 3-dimensional clipping

- 3-dimensional clipping is time consuming
- polygon division is quicker, and texture warping is also improved
- use the scratch pad

Other optimizations

- finding bottlenecks
  - use root counter to measure calculation times
- decrease memory accesses
  (if possible, use preset data types)

Other optimizations (cont)

- use GTE macros
- if the bottleneck is in rendering, decrease the polygon count
- compiler options (-02, -03)

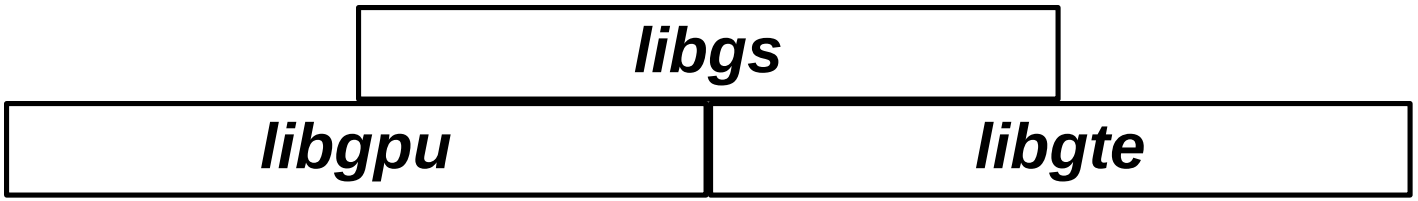
3D Graphics Sample Demonstration

libgs

What is libgs?

libgs is a graphics library constructed over libgpu, libgte

the services of libgs can be used while also using libgpu, libgte



Processing units

The processing units are as follows:

libgpu	polygon	sprite
libgte	matrix	vector
libgs	object	coordinates

Handling can be performed with more logical units

Because the handling units are large, all the know-how accumulated up to now by SCE can be utilized

Data formats

TMD format	modeling data
TIM format	image data
TOD format	motion data

Directly linked with the authoring tools

Limitations of libgs

Restrictions on freedom from use of a single paradigm, overhead

Can be partially rewritten with libgpu, libgte, which offer more freedom

libgs features

1: system based

double-buffer processing

OT control

2: coordinate system

hierarchical coordinate system

setting view point

local coordinates of object

3: object calculations

light-source calculations

perspective transformations

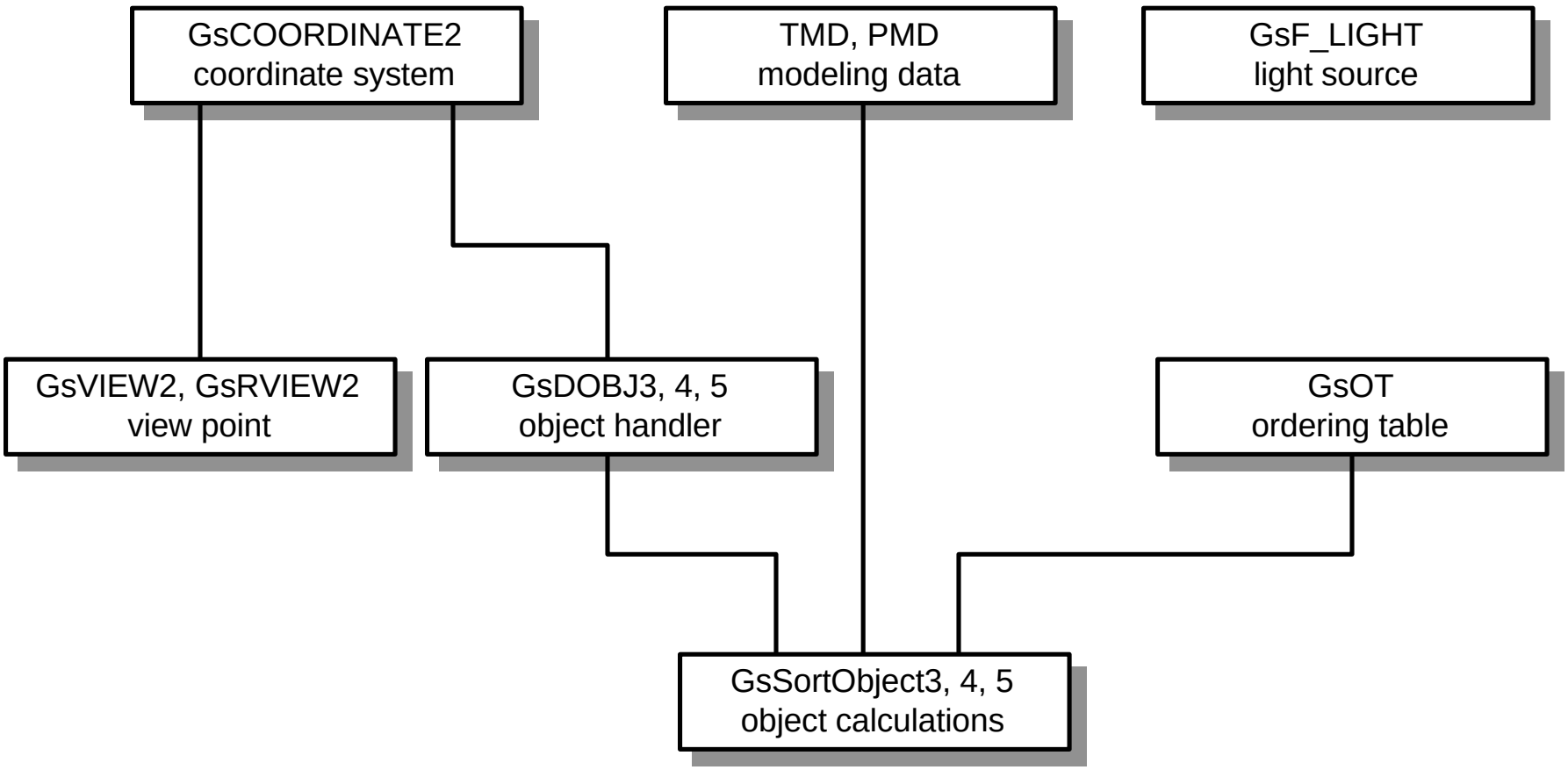
Z sorting

4: animation

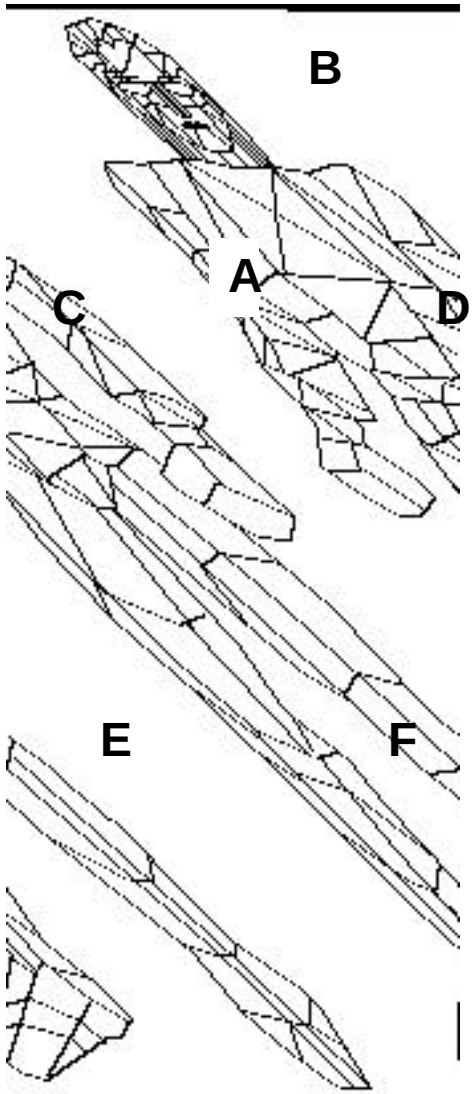
defining object motions

TOD data processing

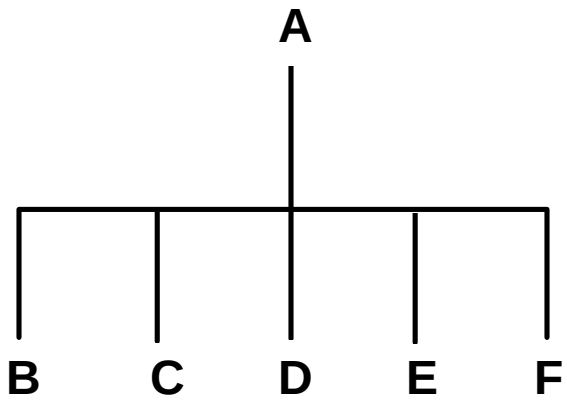
Table of interrelations of features



Coordinate system



Support for hierarchical coordinate systems



Coordinate calculations

High-speed operations using a matrix cache

a
| [A]
b
| [AB] = A'
c
| [ABC] = [A'C]
d

when deriving the c and d matrix,

normally:
c : [AB] 1
d : [ABC] 2
---

three matrix operations

with caching:
c : [AB] 1
d : [A'C] 1 as a result of A'=[AB]
---

two matrix operations

Light-source calculations

Intuitive settings based on the direction vector and the light-source color

Light-source calculations can be turned ON/OFF

FOG function

Coloration function based on ambient light

View point settings

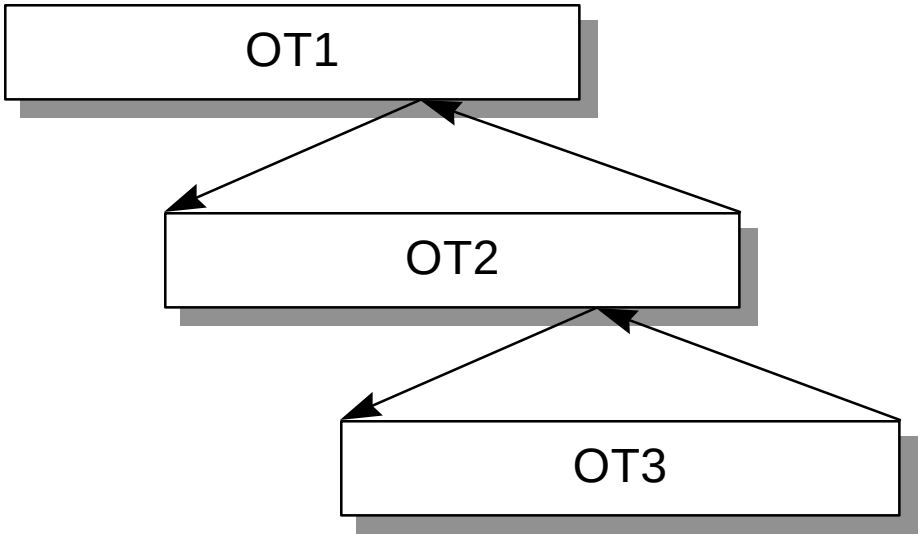
Intuitive settings based on the view point and the focus point

The view point coordinates can be set to an arbitrary coordinate, thus allowing easy implementation of first-person views

OT

Multiple OTs can be used by sorting OTs with each other

OT size can be set to variable lengthcompression of OT using OFFSET



Object calculations

Versatile object handlers that can be selected according to feature

Automatic division feature

High-speed processing routines that take full advantage of GTE speed

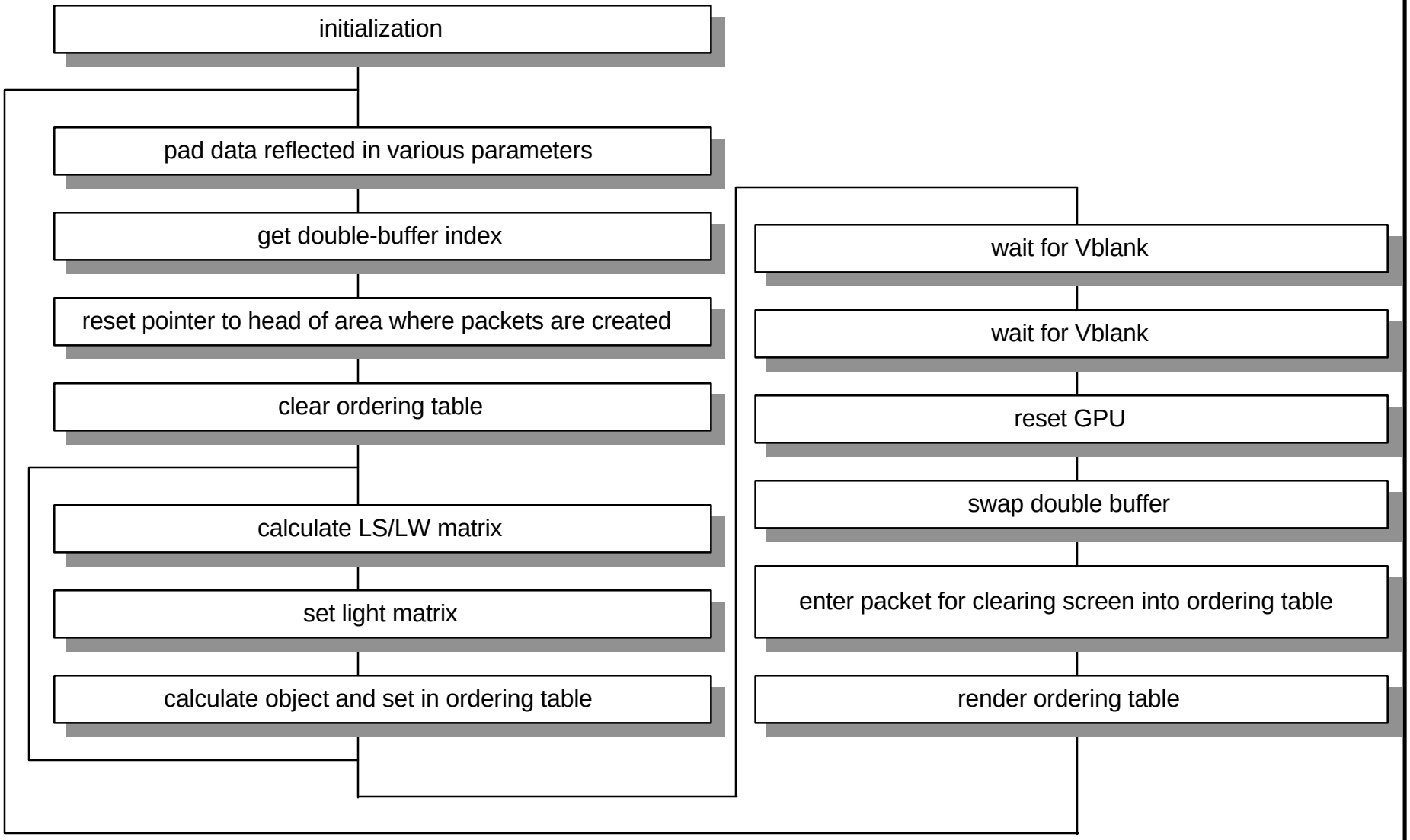
Direct processing of TMD, TOD, TIM authoring formats

Code efficiency

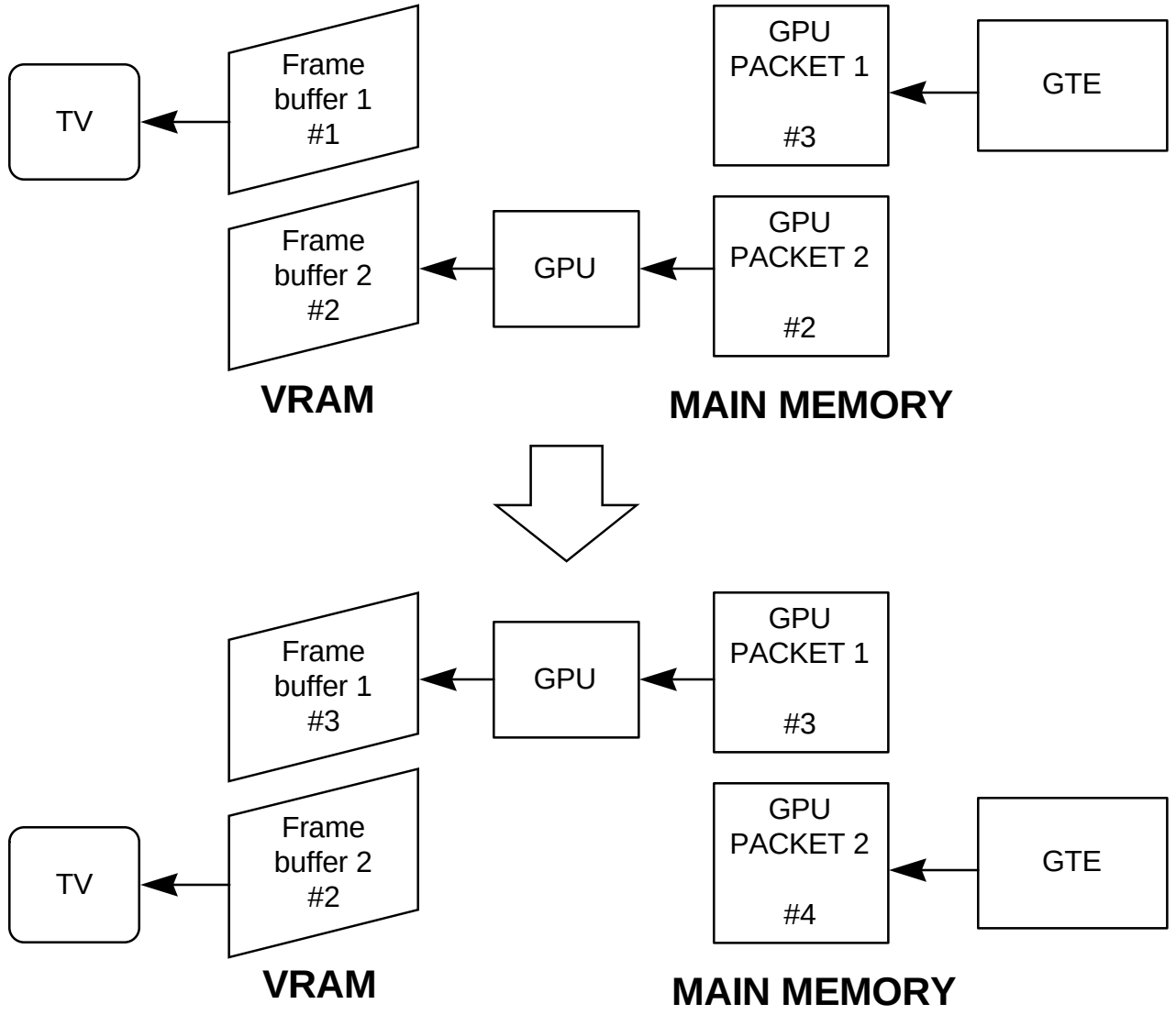
Optimization of code using dynamic linking
(linking only the functions that are used)

Explanation of sample program

Program flowchart



Frame double buffer

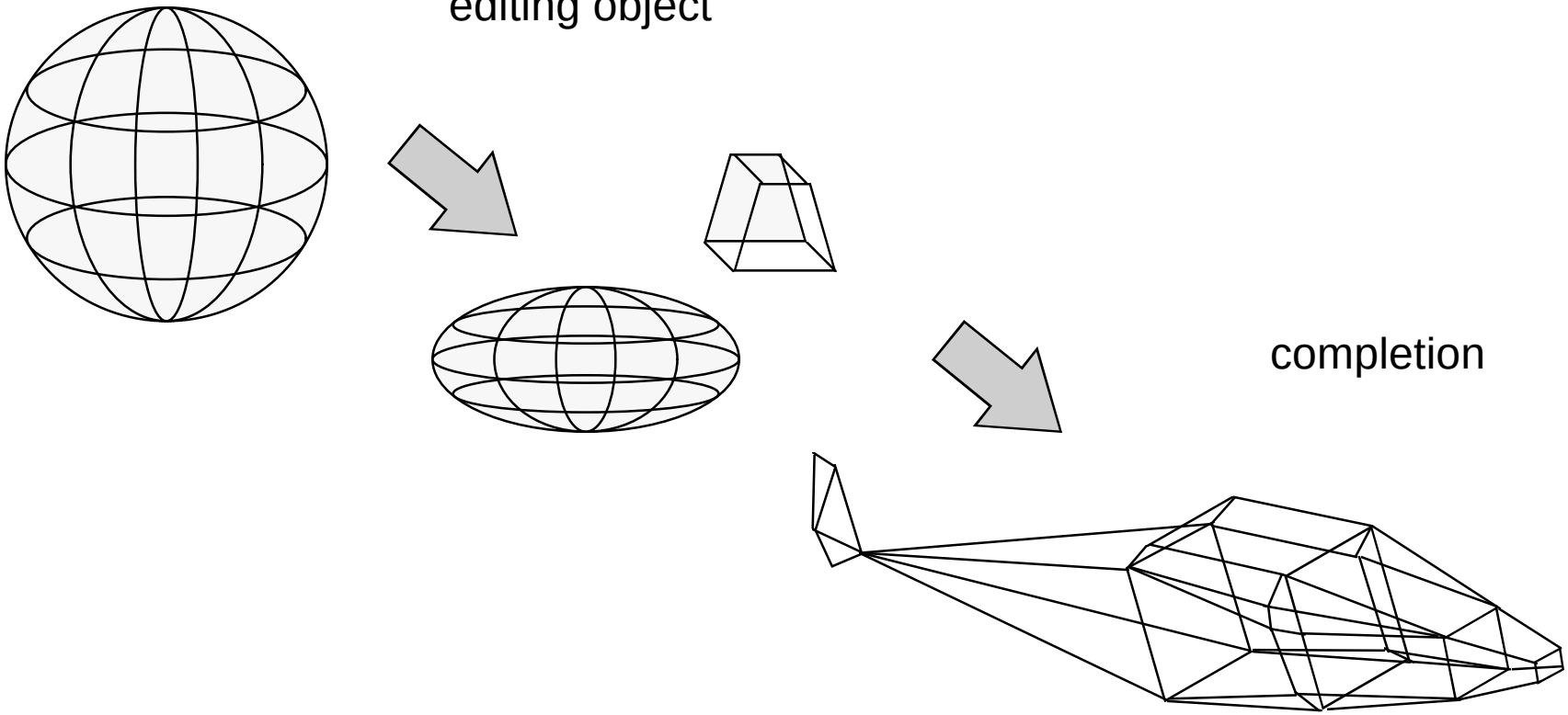


3D graphics sample demonstration

creation of 3D model

1. modeling <TrueSpace>

editing object



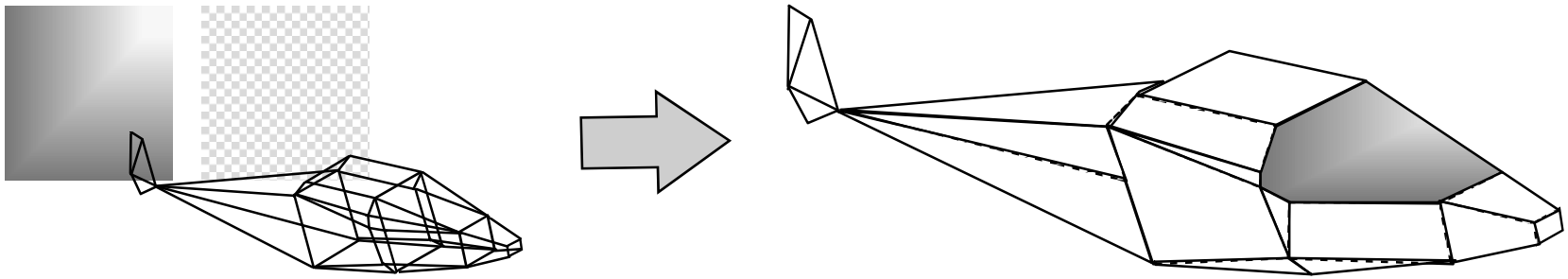
completion

3D graphics sample demonstration (cont)

2. texture mapping <TimUtil/MaterialEditor>

- set output format with TimUtil
  image origin: X=640, Y=0
  palette origin: X=0, Y=480
- activate MaterialEditor

  Select polygon and texture with "Material setting" and perform mapping



3. convert RSD data to TMD data <RSDLINK>

- convert RSD data to data that the libraries can handle (TMD) using RSDLINK