# A Low-Power Implementation of 3D Graphics System for Embedded Mobile Systems

Chanmin Park, Hyunhee Kim and Jihong Kim

School of Computer Science & Engineering,
Seoul National University, Korea

October 26, 2006
ESTIMedia 2006

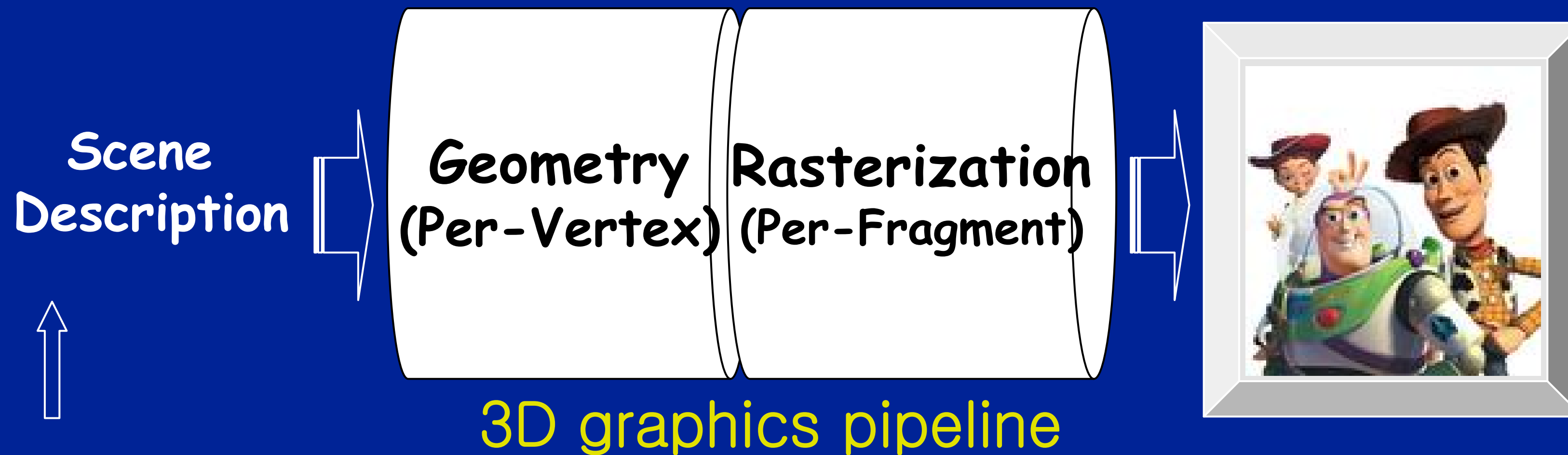
# Outline

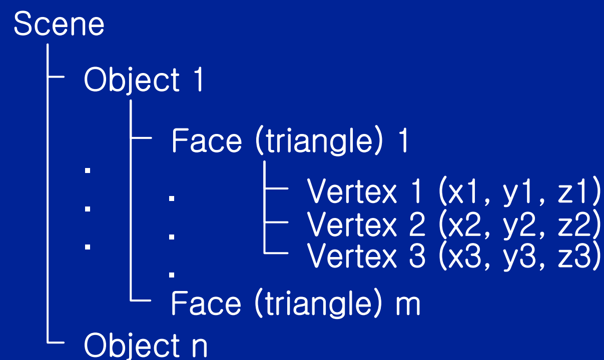
- Introduction
  - 3D Graphics 101
- Motivational examples
- Dynamic Voltage Scaling (DVS) for 3D Graphics
  - Inter-frame DVS
  - Intra-frame DVS
    - Intra-object
    - Inter-object
- Experimental Results
- Conclusions

# Introduction

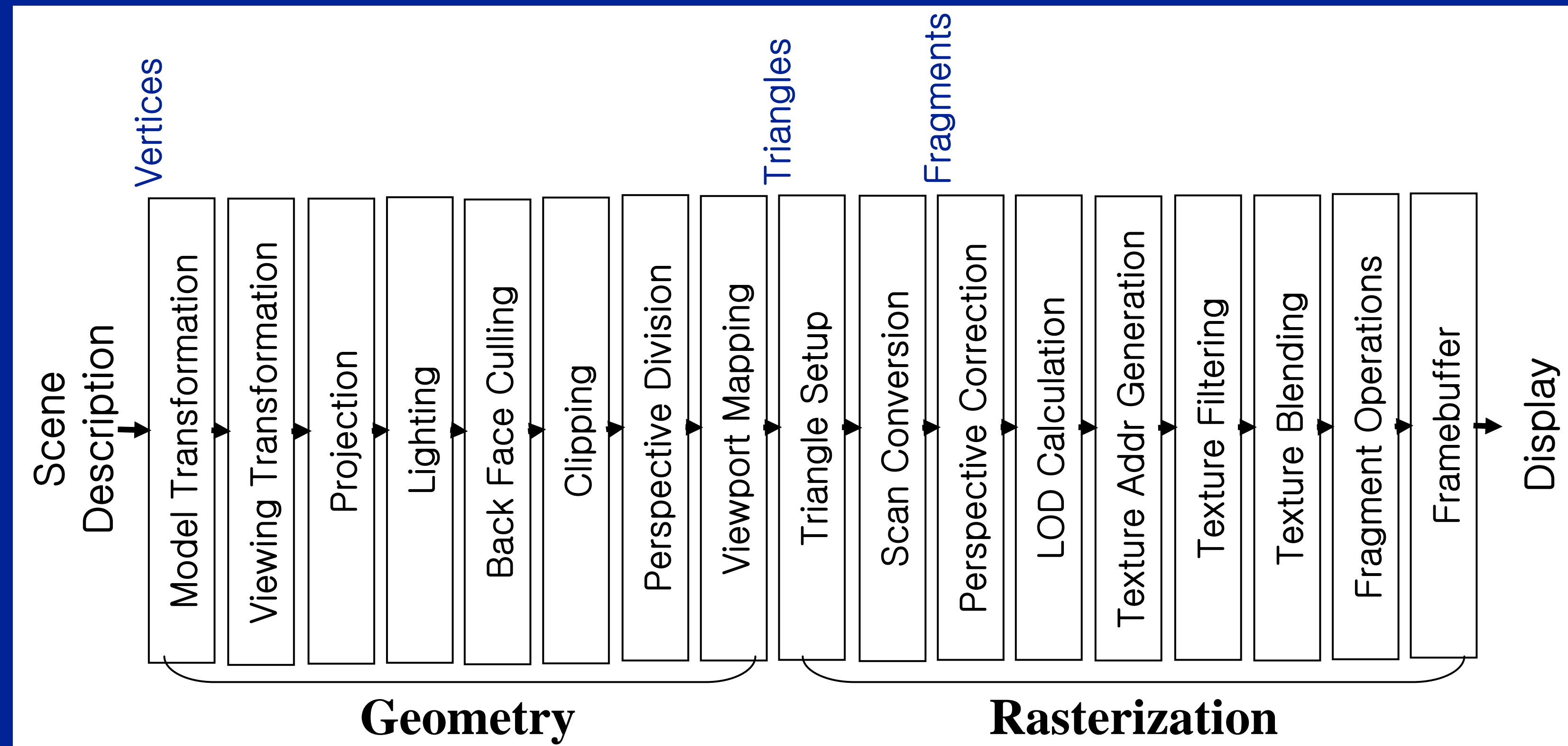
- 3D graphics became an important application for mobile devices
  - Ex) 3D games, navigation, UI, etc.
- 3D graphics applications are "**power-hungry**"
  - A large number of arithmetic operations and a high frequency of memory accesses
  - Power-aware techniques for 3D graphics are necessary
- Present a dynamic voltage scaling technique for 3D graphics

# 3D Graphics Pipeline



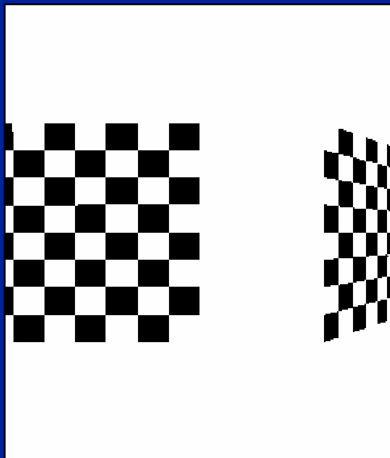
# 3D Graphics Pipeline

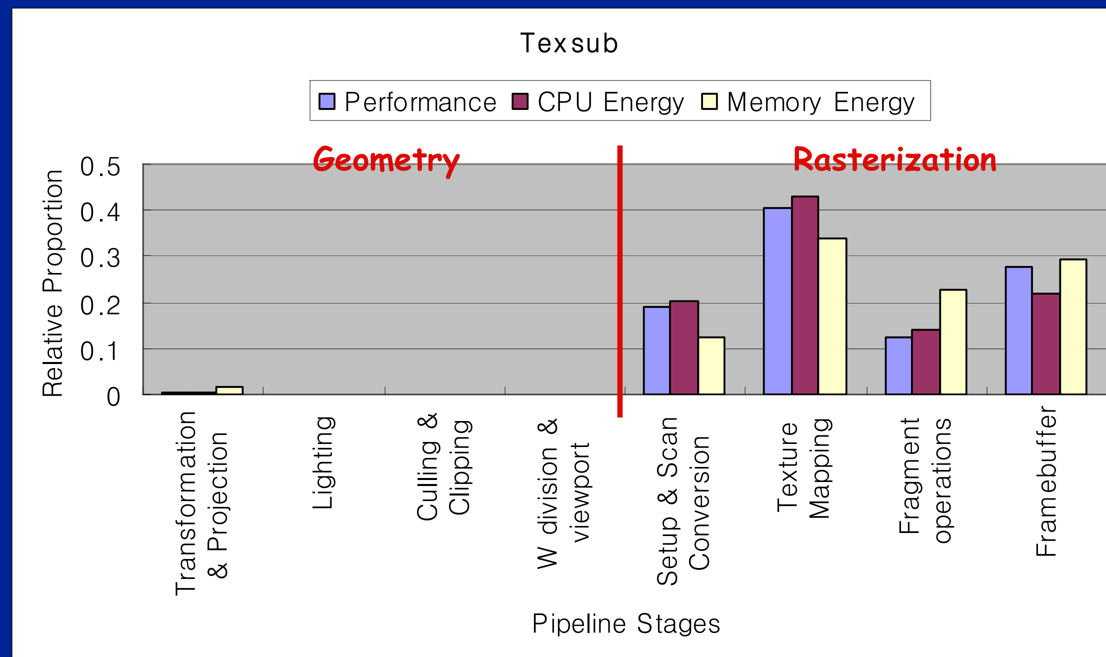



- Different applications have different processing requirements
  - Geometry-bound: a large number of vertices
  - Rasterization-bound: a large number of fragments

# Motivational Example - 1

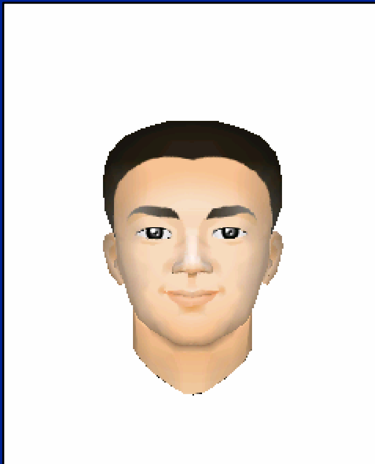
- Texsub
  - An example of OpenGL tutorial
  - 8 vertices, 24388 fragments
  - Consumes most of energy in Rasterization phase

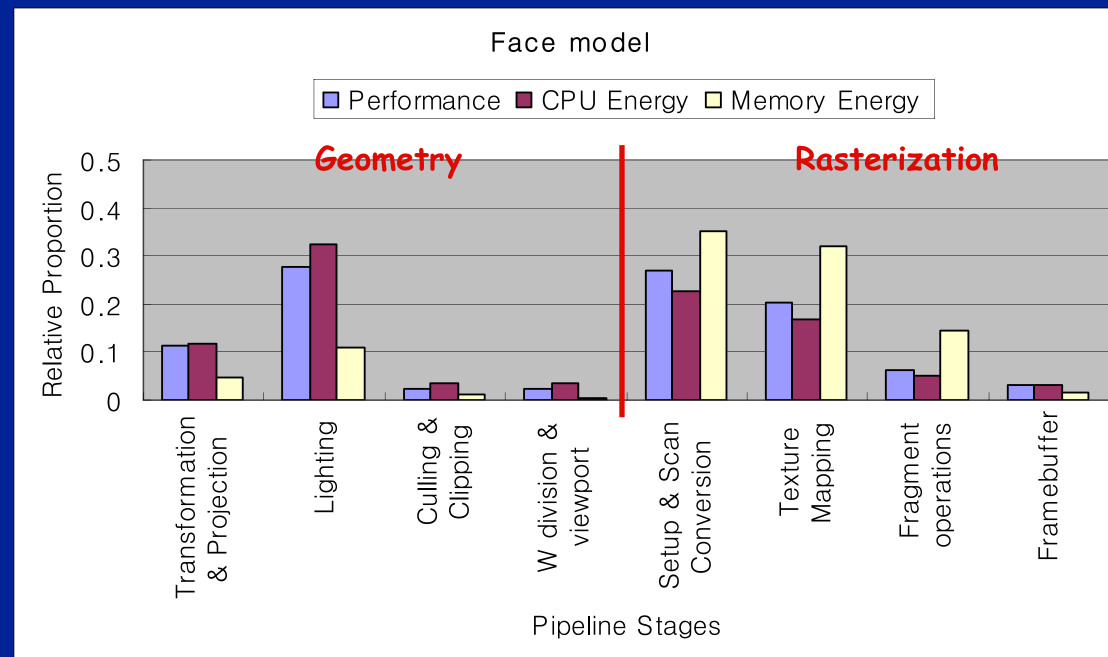




# Motivational Example - 2

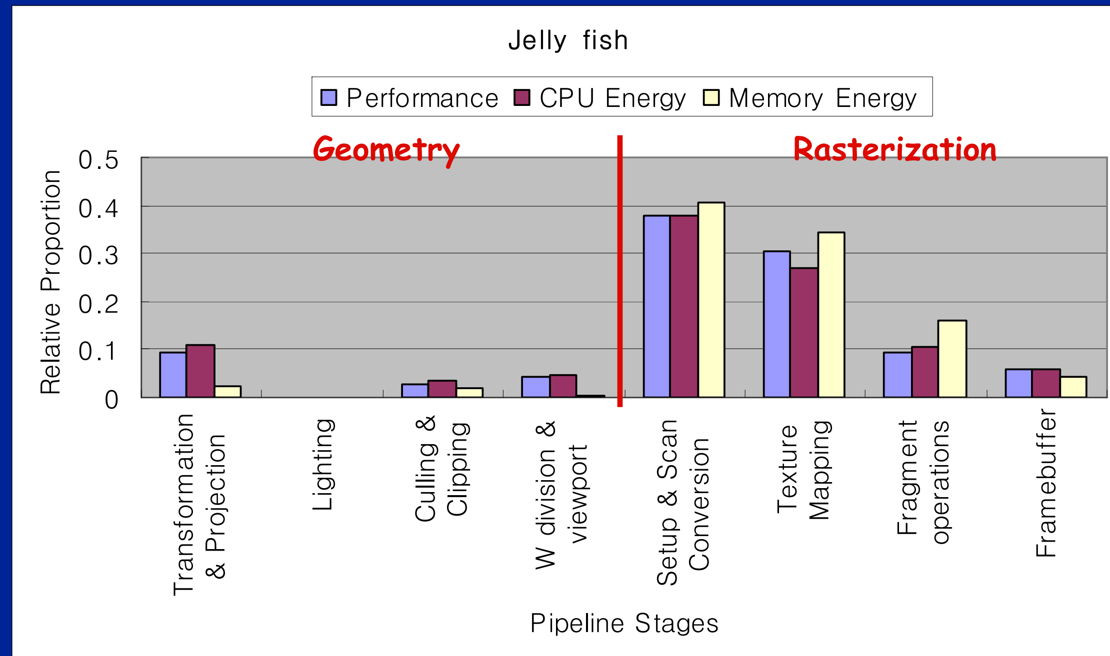
- Face model
  - A 3D character model
  - 4281 vertices, 16562 fragments, lighting
  - Consumes 52% of energy in Geometry phase



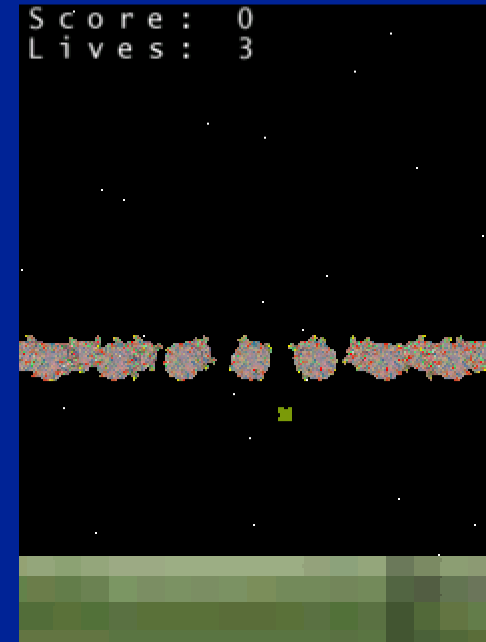


# Motivational Example - 3

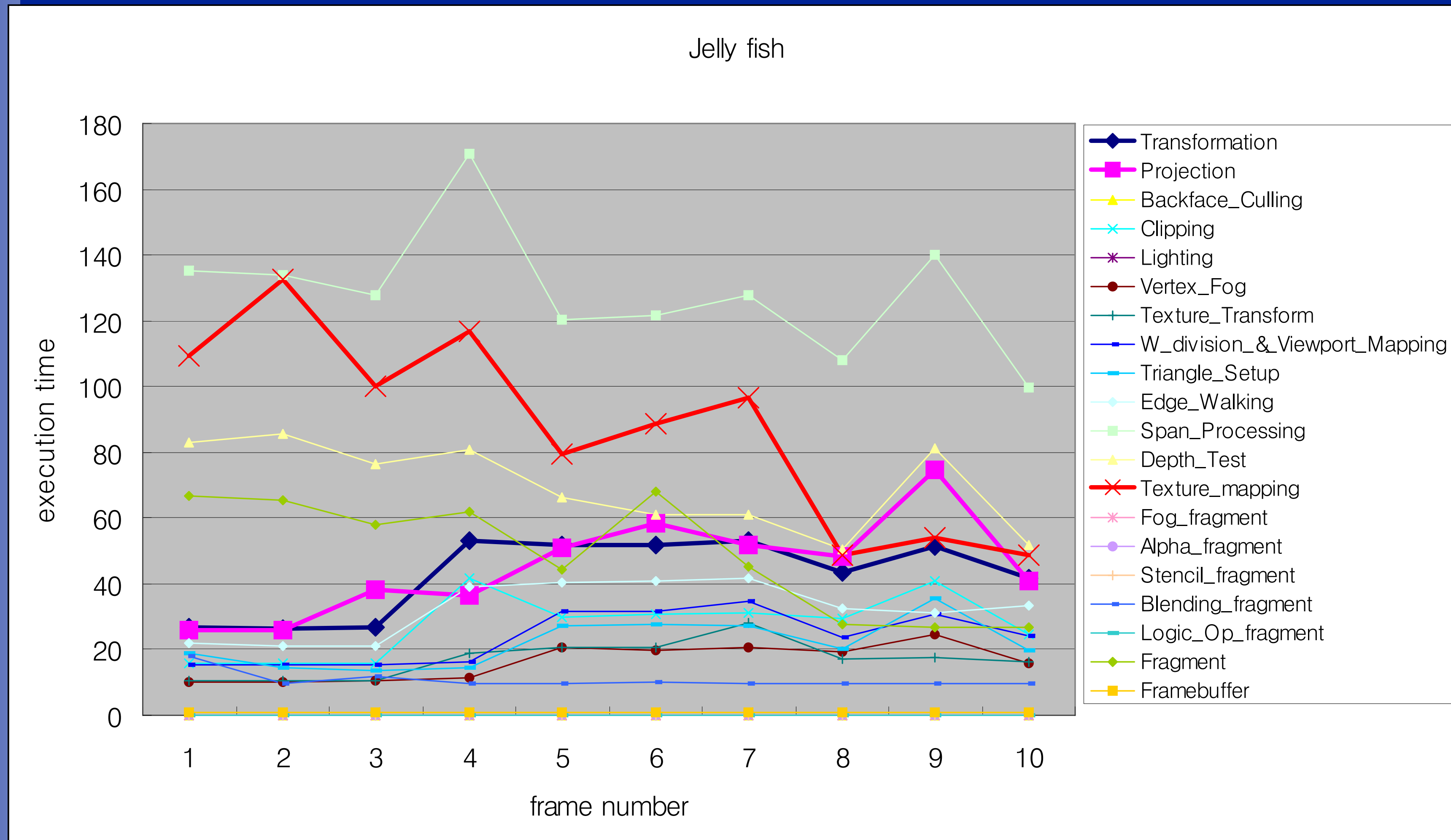
- Jelly fish
  - A shooting game
  - 9187 vertices, 47070 fragments on average
  - Dynamically changing workloads due to moving objects & camera movements



# Moving Objects & Camera View Point Variations along frames

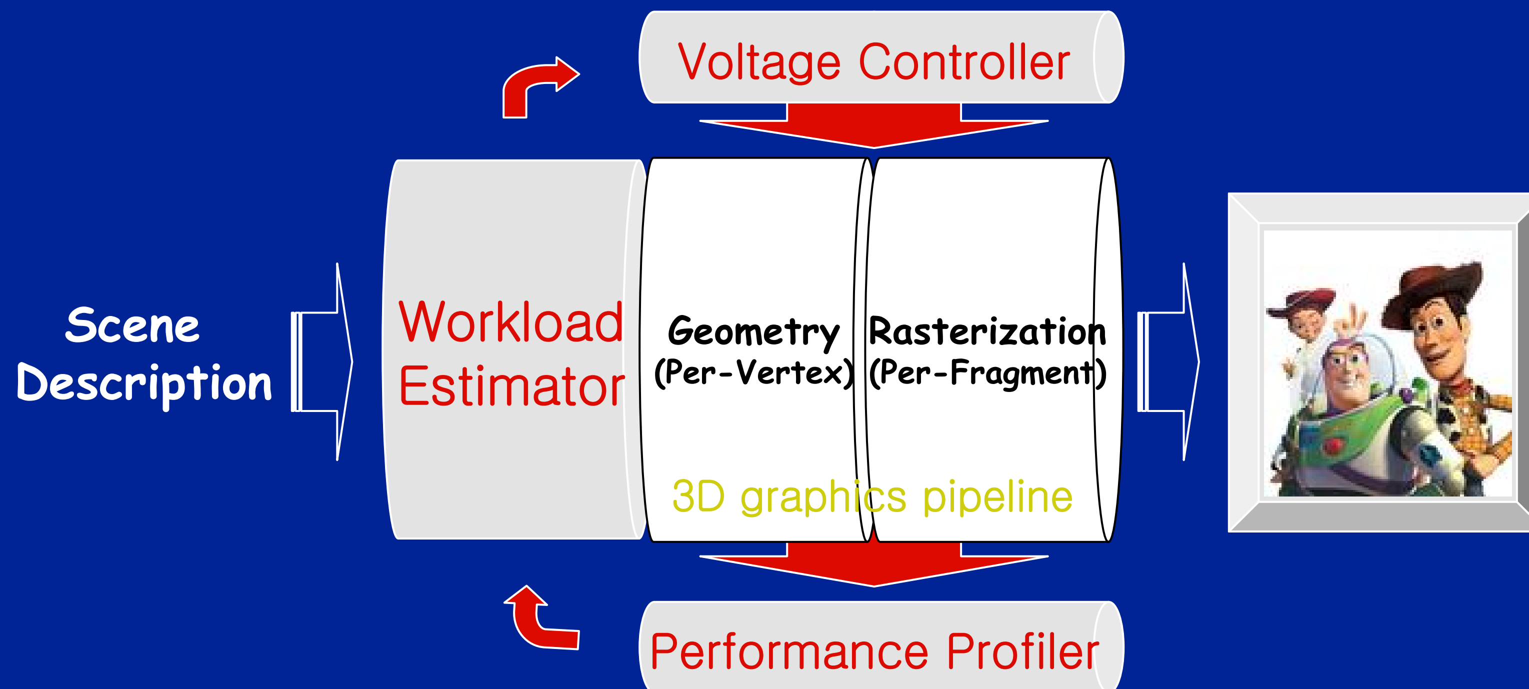


# Workload variation



# DVS for Low-Power 3D Graphics

- Key steps for DVS
  - Detection of slack intervals based on workloads
  - Voltage scaling policy for slack intervals
- Conceptual Diagram



# Workload Estimator

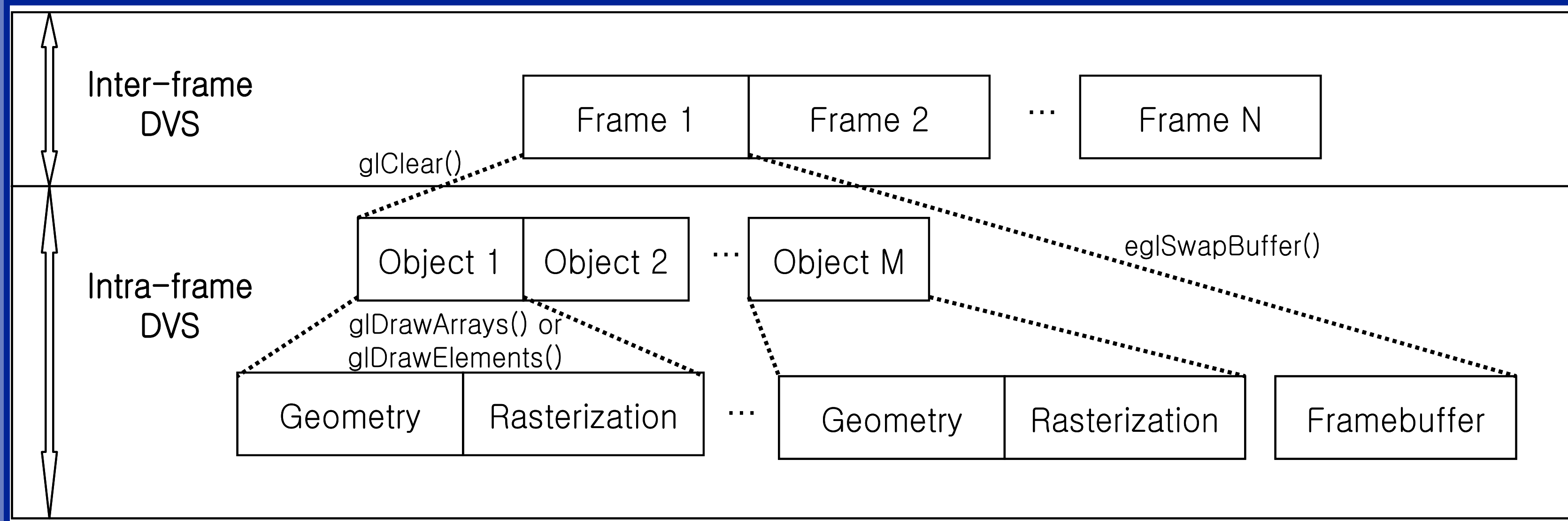
- 1. Slack Identification
- 2. Slack Distribution
- 3. Frequency & voltage level Decision

- For the first frame, an object list is created
  - For each object, we store information
    - the number of vertices, the number of triangles, the number of fragments, execution time, lighting parameters, texture parameters for each object

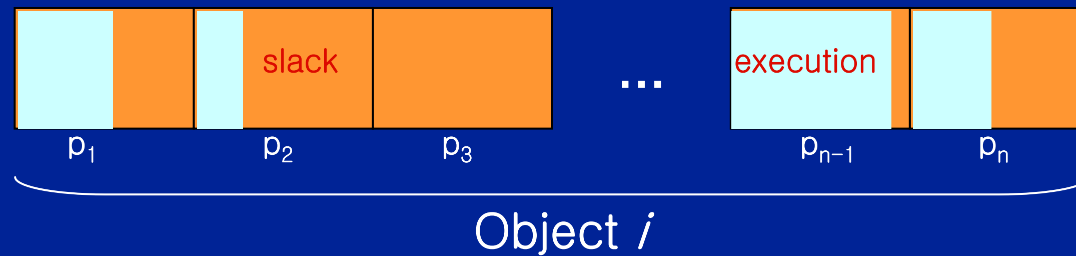
- For every frame completed
  - Object variations are updated
  - Error tolerance threshold : T
    - Validates estimation and controls the error tolerance
  - If the variation is larger than T, the object list is reset

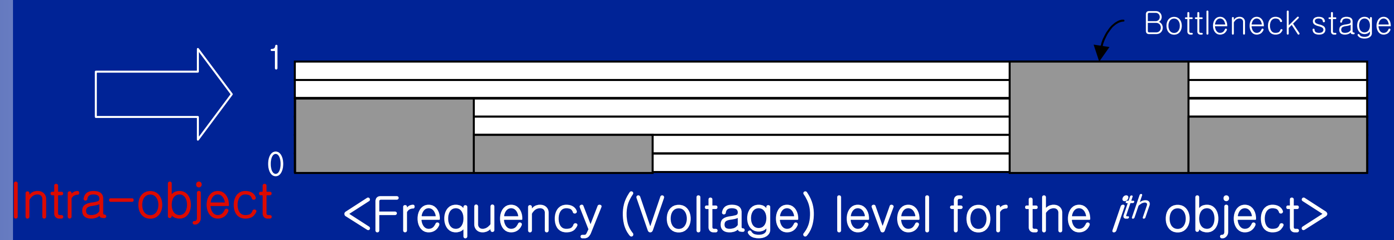
# Two Layers of Slack Identification

- Inter-frame DVS
  - The voltage is adjusted by a frame granularity based on the slack times generated from the previous frame
- Intra-frame DVS
  - The voltage is adjusted by an object granularity within a frame
  - Intra-object & Inter-object based on slack distribution

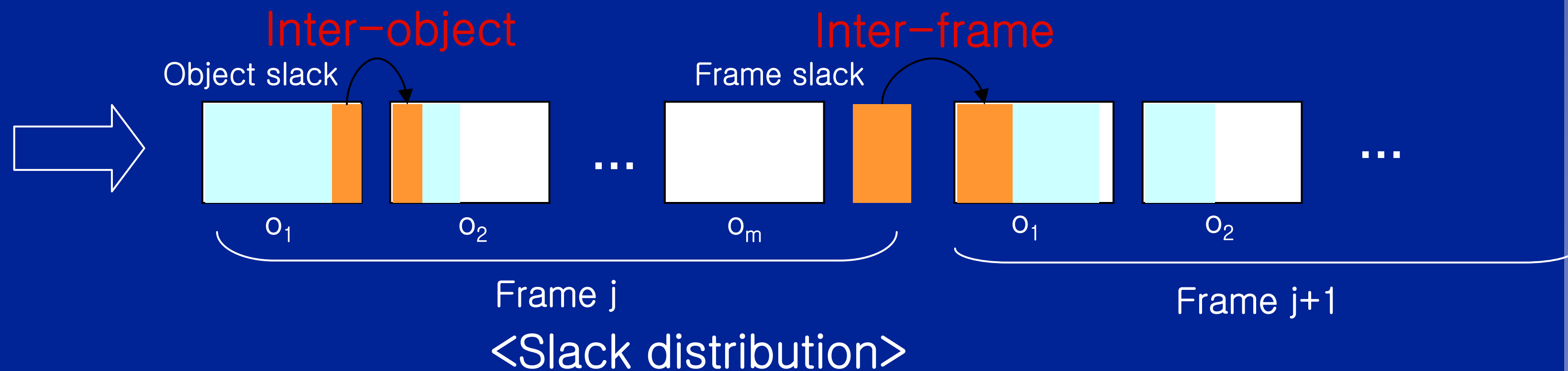


# Greedy Slack Distribution




<Frequency (Voltage) level for the $i^{th}$ object>




<Slack distribution>

# To get more slacks

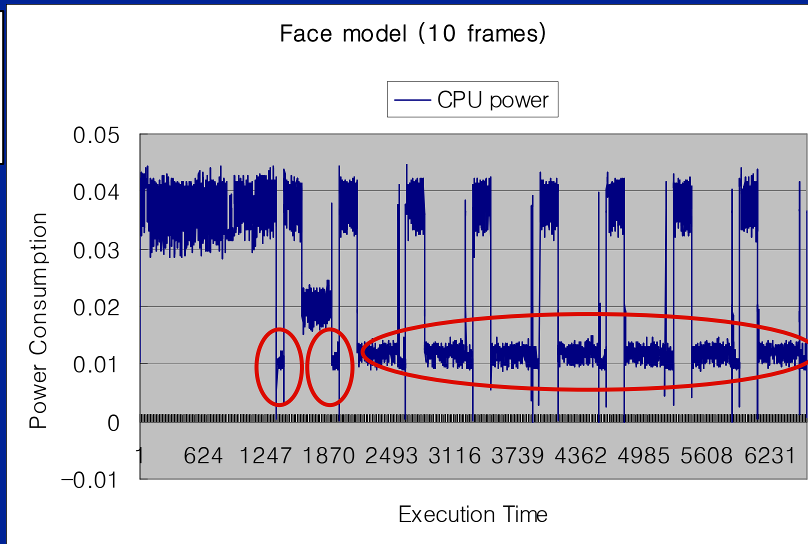
- Vertex caching
  - Avoid repetitive transformation and lighting calculations of the same (shared) vertices to get more slacks

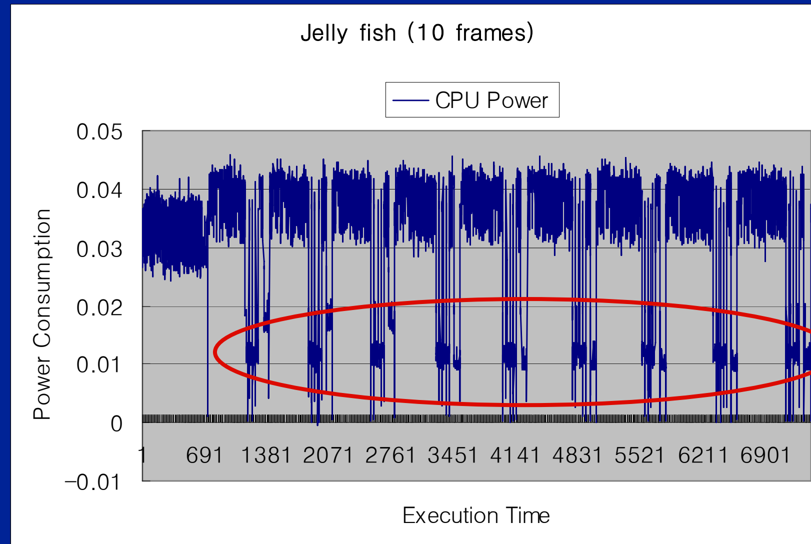
# Experimental Results

- Used a DVS-aware PDA development board
  - Processor : Intel Xscale PXA255
    - Frequency: 7 levels between 100Mhz and 400Mhz
    - Voltage: 3 levels between 1.0V and 1.3V
  - RAM: 64MB
  - Cache: 32 KB I$ & D$
  - Display: 240*320, 16bit color, QVGA
  - OS: Embedded Linux (ver. 2.4.19)

- Power measurements done using DAQ
- OpenGL ES 1.1 based power-aware 3D library
- Test apps: Redbook samples, Facemodel, Jellyfish

# Voltage Scaling Patterns



- Face model benefited from the vertex caching
- Less opportunities for voltage scaling in Geometry phase due to many short slacks

  →most voltage scalings occur in Rasterization

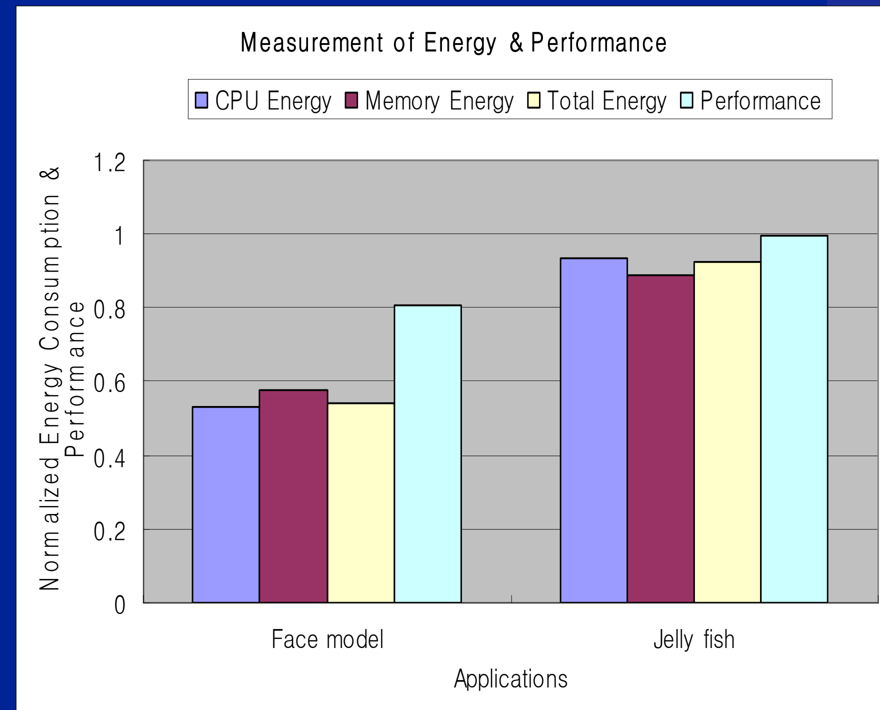
# Energy Consumption Comparison

- Face model

  - 47% energy saving in CPU & 43% energy saving in memory
  - 46% energy saving for total energy consumption

- Jellyfish

  - 8% energy saving for total energy consumption



# Conclusions

- Described a DVS scheme applicable to 3D graphics
  - Intra-frame DVS & Inter-frame DVS based on the application's varying workloads
- Implemented the proposed technique using OpenGL ES 1.1
- Achieved an energy saving of up to 46% over a power-unaware implementation

For more information, cmpark@davinci.snu.ac.kr

Thank you

# Appendix

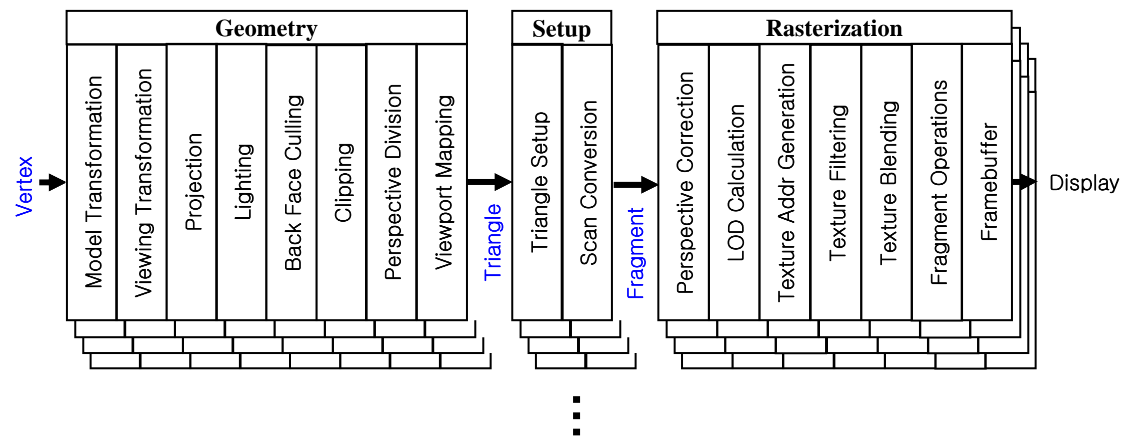
# System Model – parameters

- Execution time of the $j^{th}$ frame and bottleneck stage, for each $i^{th}$ pipeline stage,
  - $C_i$ : WCET of $i^{th}$ pipeline stage at the maximum processor speed
  - $S_i$ : state enabled or disabled by graphics feature
    - Ex) glEnable(GL_LIGHT), glEnable(GL_TEXTURE_2D), glEnable(GL_DEPTH_TEST)
  - $N_i$ : the iteration factor
    - Ex) number of vertices, number of triangles, number of fragments, etc.
  - $P_{th-i}$ : the throughput factor
    - Ex) 128 bit SIMD → (x, y, z, w) / cycle(s), 4 colors / cycle(s), etc.
      8 Pixel Processing Unit → 8 frgments / cycle(s), etc.

$$D_j = \sum \frac{C_i S_i N_i}{P_{th-i}}$$

$$B_j = \max \left\{ \frac{C_i S_i N_i}{P_{th-i}} \right\}$$

$$(1 \le i \le n)$$



# Intra-Frame DVS

- Restate the execution time, when a scene has *m* objects

$$D_j = \sum_o \sum_i \frac{C_i S_i^o N_i^o}{P_{th-i}} \quad (1 \le i \le n,\ 1 \le o \le m)$$

- Static slacks due to bottleneck stages
  - Frequency setting: $F_i^o$

$$F_o^{static} = \frac{\sum_{i=1}^{n} \frac{C_i S_i^o N_i^o}{P_{th-i}}}{\sum_{i=1}^{n} \frac{C_i S_{all}^o N_i^o}{P_{th-i}}} \qquad F_i^o = F_o^{static} \cdot \frac{\left(\frac{C_i S_i N_i}{P_{th-i}}\right)}{B_j}$$

where $S_{all}^o$ means all features $S_i^o$ are enabled,

# Intra-Frame DVS

- Dynamic slacks between objects
  - Using slacks from the previous object
  - Compensating the misprediction in the previous stage
  - Frequency setting:

$$F_{o+1}^{dynamic} = \frac{\displaystyle\sum_{j=o+1}^{m}\sum_{i=1}^{n}\frac{C_i S_i^j N_i^j}{P_{th-i}}}{\displaystyle\sum_{j=1}^{m}\sum_{i=1}^{n}\frac{C_i S_{all}^j N_i^j}{P_{th-i}} - \sum_{j=1}^{o}\sum_{i=1}^{n}\frac{C_i S_i^j N_i^j}{P_{th-i}}}$$

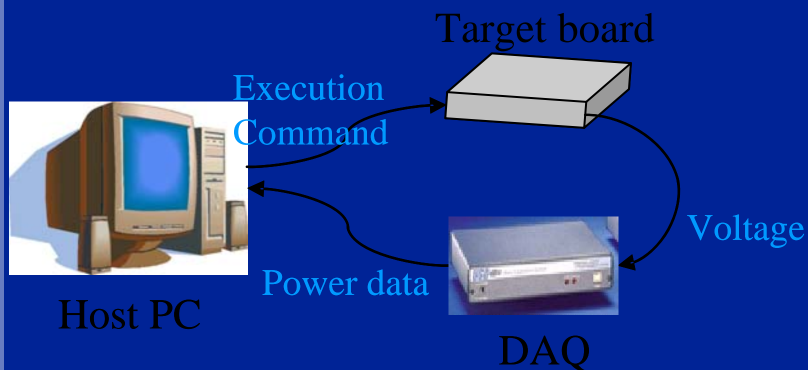
where $S_{all}^j$ means $j^{th}$ object has all enabled features

# Inter-Frame DVS

- Dynamic slacks between frames
  - The slack from the previous frame is used by the first object in the next frame
  - Frequency setting: the same as in the case of dynamic slacks between objects
- Compensating the misprediction in the previous frame
  - Since the intra-frame estimation is a conservative approach, it cannot find all the slack times in advance
  - Such unused dynamic slacks are added to the deadline for the next frame
- If the frame rate is controlled by an application itself, however, the *inter-frame* DVS has no effect on having slack time, since we cannot start processing the next scene earlier at the level of library

# Measurement of Energy Consumption

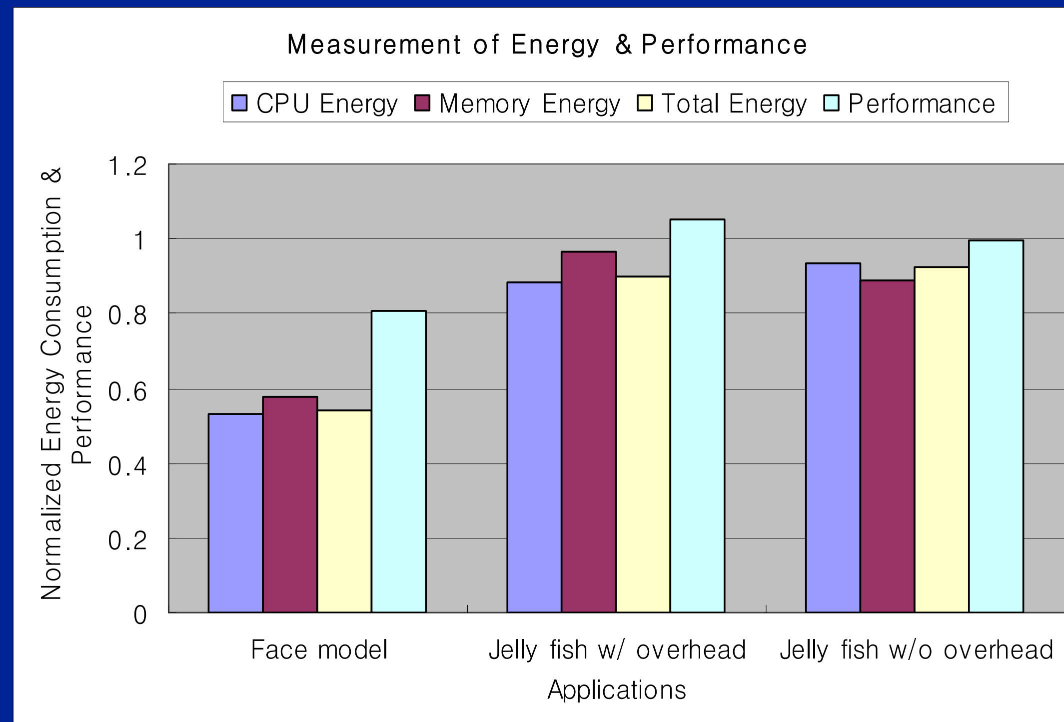
- Physical measurement
  - Host PC : measuring the power consumption of CPU and memory separately (through separate probes)
  - DAQ/Labview: power measuring device




<Fig. Environment for Physical Measurement>

<Fig. Target Board>

# Experimental Results




<Fig. Experimental results>

# End

# System Model

- **The execution time of the $j^{th}$ frame**

$$D_j = \sum \frac{C_i S_i N_i}{P_{th-i}} \qquad (1 \le i \le n)$$

For each $i^{th}$ pipeline stage,

$C_i$ : WCET of $i^{th}$ pipeline stage at the maximum processor speed

$S_i$ : state enabled or disabled by graphics feature

$N_i$ : the iteration factor

$P_{th-i}$ : the throughput factor

- **The execution time of bottleneck stage**

$$B_j = \max\left\{ \frac{C_i S_i N_i}{P_{th-i}} \right\} \qquad (1 \le i \le n)$$

# Intra-Frame DVS

- Restate the execution time, when a scene has *m* objects

$$D_j = \sum_o \sum_i \frac{C_i S_i^o N_i^o}{P_{th-i}} \quad (1 \le i \le n,\ 1 \le o \le m)$$

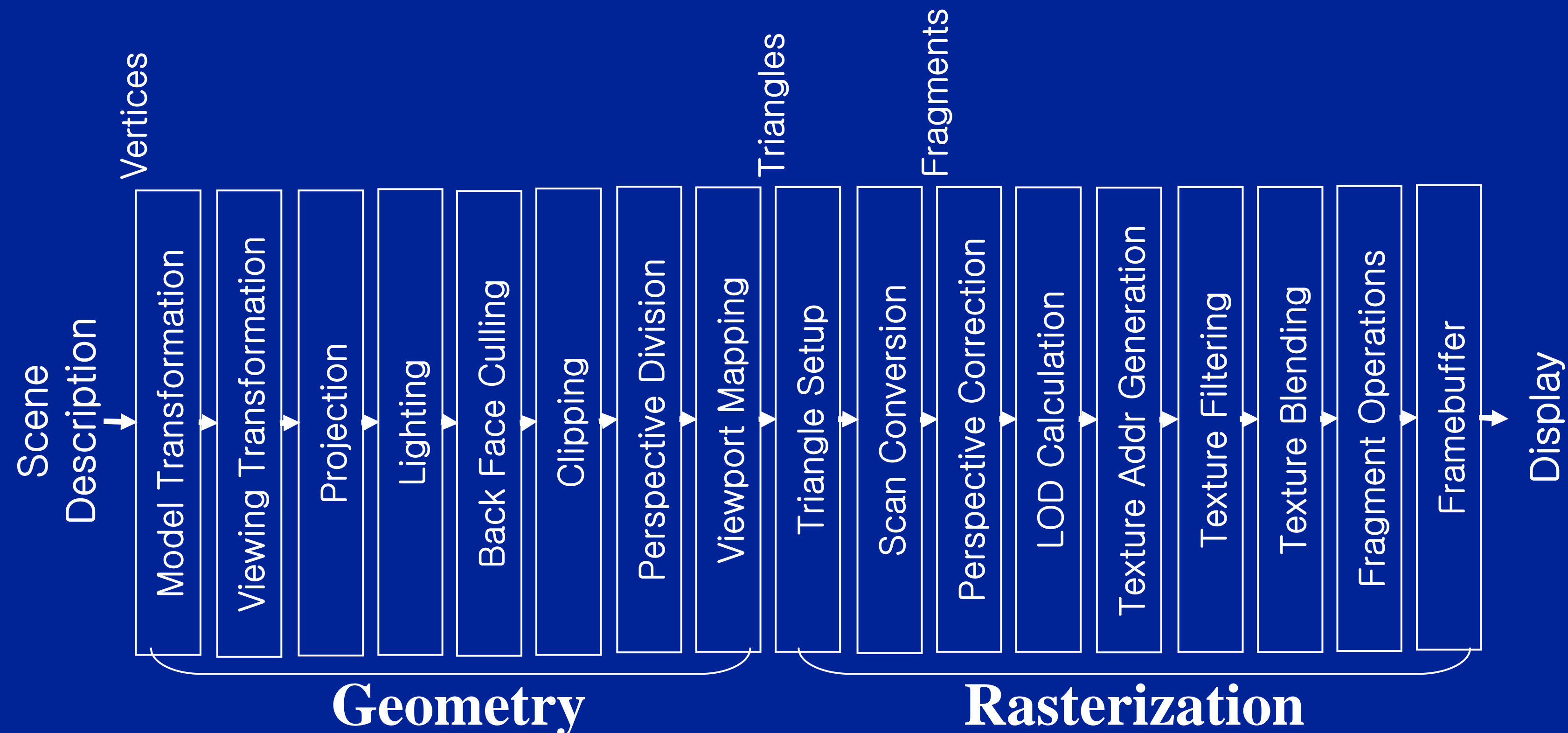
- Static slacks due to bottleneck stages
  - Frequency setting: $F_i^o$

$$F_o^{static} = \frac{\sum_{i=1}^{n} \frac{C_i S_i^o N_i^o}{P_{th-i}}}{\sum_{i=1}^{n} \frac{C_i S_{all}^o N_i^o}{P_{th-i}}}$$ where $S_{all}^o$ means all features $S_i^o$ are enabled, $$F_i^o = F_o^{static} \cdot \frac{\left(\frac{C_i S_i N_i}{P_{th-i}}\right)}{B_j}$$

- Dynamic slacks between objects
  - Using slacks from the previous object
  - Compensating the misprediction in the previous stage
  - Frequency setting:

$$F_{o+1}^{dynamic} = \frac{\sum_{j=o+1}^{m} \sum_{i=1}^{n} \frac{C_i S_i^j N_i^j}{P_{th-i}}}{\sum_{j=1}^{m} \sum_{i=1}^{n} \frac{C_i S_{all}^j N_i^j}{P_{th-i}} - \sum_{j=1}^{o} \sum_{i=1}^{n} \frac{C_i S_i^j N_i^j}{P_{th-i}}}$$ where $S_{all}^j$ means $j^{th}$ object has all enabled features

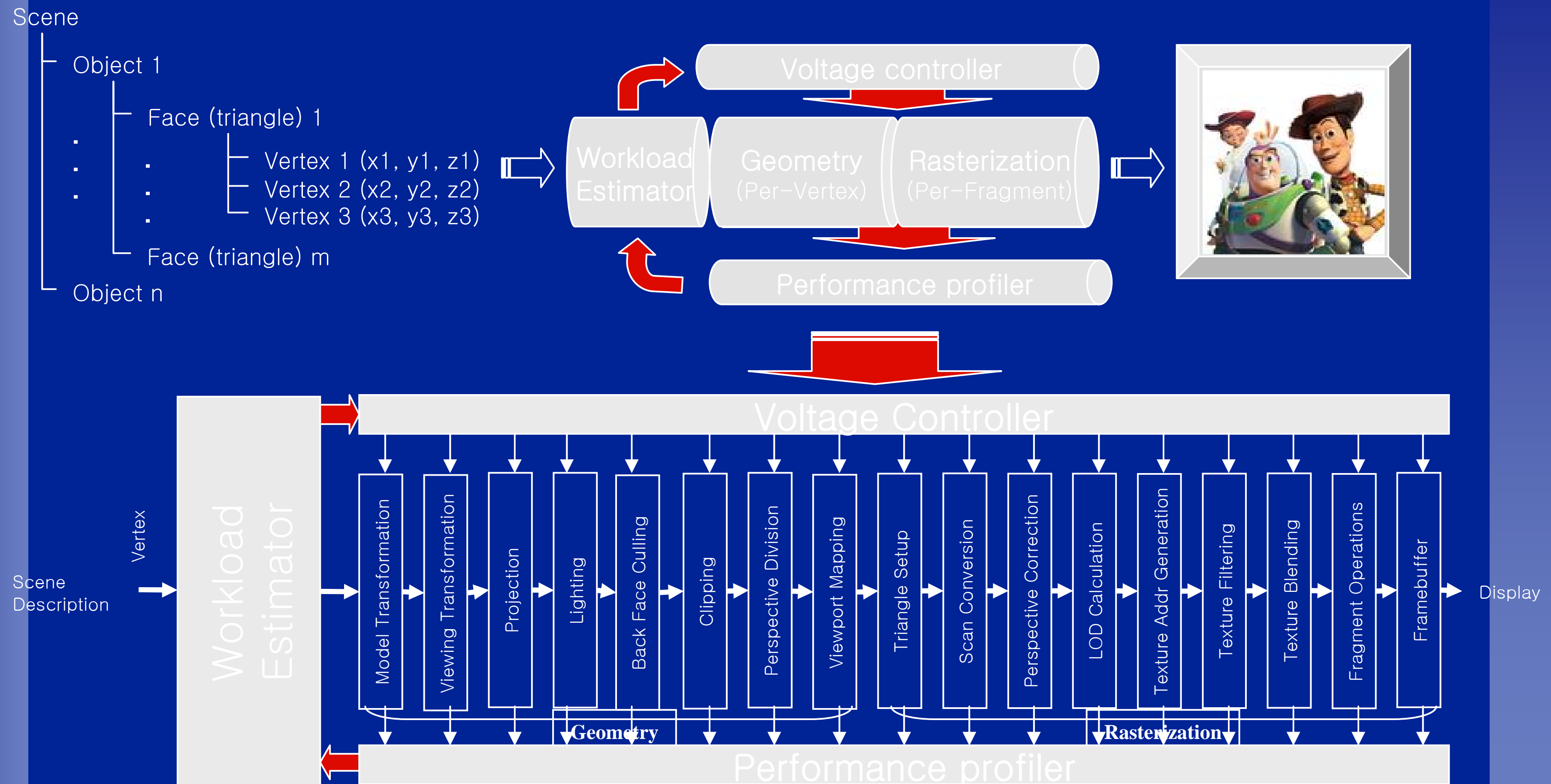
# 3D Graphics Pipeline




- Different applications have different processing requirements
  - Geometry-bound: a large number of vertices
  - Rasterization-bound: a large number of fragments

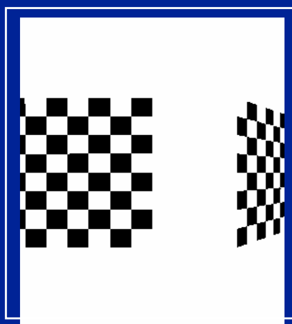
# DVS for Low-Power 3D Graphics

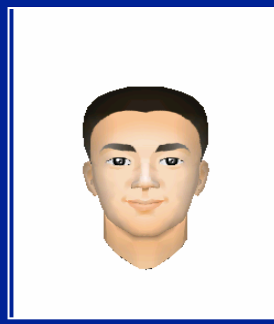
- **Conceptual Diagram**



# Motivational Examples

- **Application Characteristics**
  - Geometry-limited
  - Fill (Rasterization)-limited


a. Texsub

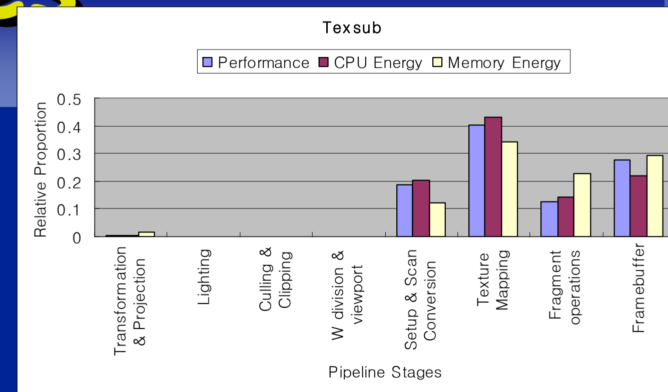

b. Face model

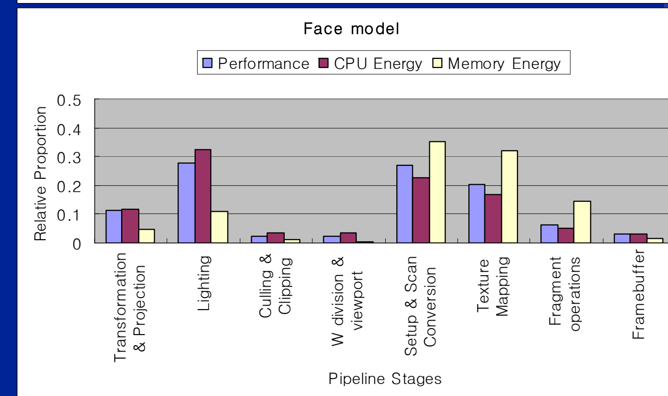
c. Jellyfish

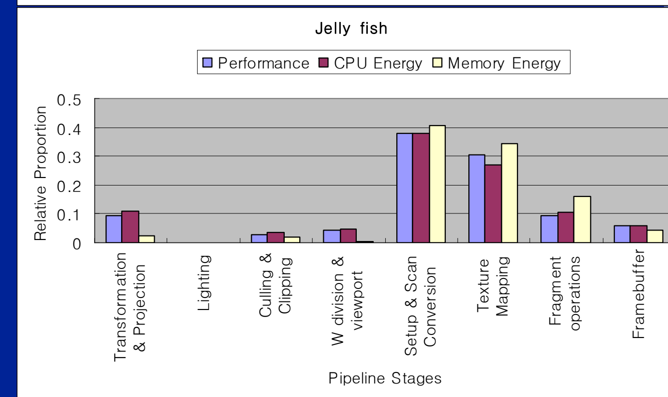
<Fig. Applications>

<Table. The statistics of applications features>

| Application | Vertex | Triangle | Fragment | Texel access | Time(sec) | Lighting |
|---|---|---|---|---|---|---|
| Texsub | 8 | 4 | 24388 | 24388 | 0.161571 | x |
| Face model | 4281 | 1427 | 16562 | 16562 | 0.806431 | 0 |
| Jellyfish (average) | 9187 | 3073 | 47070 | 47006 | 0.669926 | x |




<Fig. Performance & Energy>

# Workload variation – moving objects