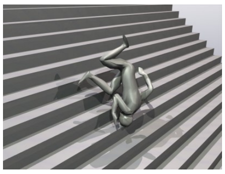
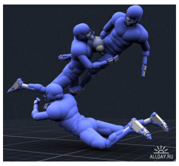
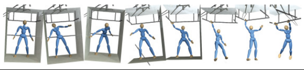




# Animation de personnage : motion capture et animation physique

Alexandre Meyer
M2pro

1

## CharAnimation : mocap VS physique

- Mocap
  - \+ Réalisme
  - \- Réalisme pour 1 animation
  - \- Système lourd
  - \- Edition fastidieuse (souvent intervention humaine)
- Animation physique
  - +Tout automatique
  - \- Tout automatique

➔ Mélanger les deux!!



## (Parenthèse : physique -cf. cours Fzara-

- Particule = 1 point animé
  - P = position = (X(t), Y(t), Z(t)) = ($X_t$ , $Y_t$ , $Z_t$ )
    au temps précédent ($X_{t-\Delta t}$ , $Y_{t-\Delta t}$ , $Z_{t-\Delta t}$ )
    sous entendu temps_suivant = t+Δt
  - V = vitesse = dP/dt

$$V_t = \frac{dP}{dt} = \lim_{\Delta t \to 0} \frac{\Delta P}{\Delta t} \approx \frac{P_t - P_{t-\Delta t}}{\Delta t}$$



## Euler explicite

- Particule = 1 point animé
  - A = accélération = dV/dt

$$A_t = \frac{dV}{dt} = \lim_{\Delta t \to 0} \frac{\Delta V}{\Delta t}$$
$$\approx \frac{V_t - V_{t-\Delta t}}{\Delta t}$$
$$\approx \frac{P_t - P_{t-\Delta t} - (P_{t-\Delta t} - P_{t-2\Delta t})}{\Delta t^2}$$
$$\approx \frac{P_t - 2P_{t-\Delta t} + P_{t-2\Delta t}}{\Delta t^2}$$



## Euler explicite

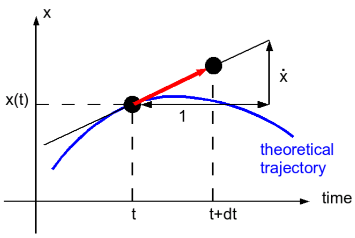
- En connaissant la position au 2 temps précédents,

$$\sum F = F = mA$$
$$A_t \approx \frac{P_t - 2P_{t-\Delta t} + P_{t-2\Delta t}}{\Delta t^2}$$
$$\boxed{donc \Rightarrow P_t = \Delta t^2 \times \frac{F}{m} + 2P_{t-\Delta t} - P_{t-2\Delta t}}$$



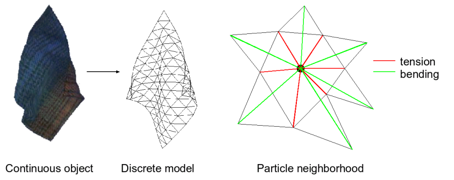
## Euler explicite

- Approximation de A et V
  - Mise à jour suit la tangente
  - Pour être stable Δt doit être petit



## Système masse-ressort

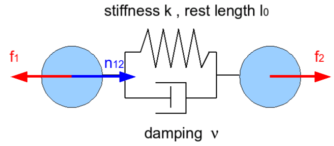
- Sommet = particles, arêtes = ressort




- Simple mais paramètres difficiles à régler



## Système masse-ressort



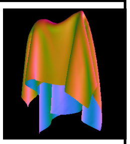

- Force viscoelastique (k=raideur du ressort)
  - Force de rappel : $F = -k \cdot (l - l_0)$
  - F Viscosité

$$f_1 = \left(k\frac{\|\mathbf{q}_2-\mathbf{q}_1\|-l_0}{l_0} + \nu(\dot{\mathbf{q}}_2-\dot{\mathbf{q}}_1).\mathbf{n}_{12}\right)\mathbf{n}_{12}$$

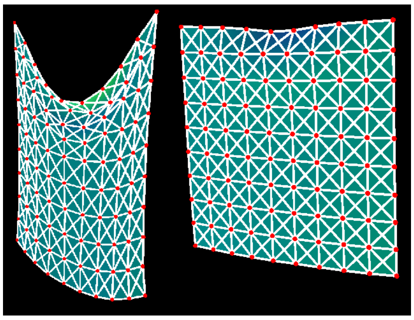
$$\mathbf{n}_{12} = \frac{\mathbf{q}_2-\mathbf{q}_1}{\|\mathbf{q}_2-\mathbf{q}_1\|}$$



## Exemple d'utilisation : tissu



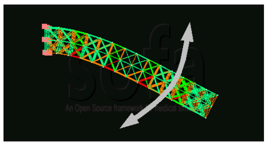
- Maillage de masse-ressort

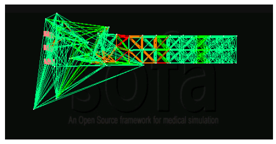




## Système masse-ressort

- Raideur du ressort : k
  - $F = -k \cdot (l - l_0)$
- K petit (objet « mou »)
- K élevé (objet rigide) : stabilité demande dt petit sinon le système explose



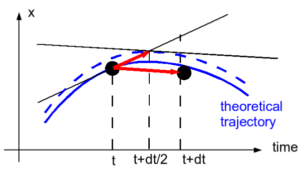




## Schéma d'intégration

- Runge-Kutta
  - Calcul un pas de temps dt/2 fictif avec Euler
  - Calcul la dérivée
  - Utilise cette dérivée pour un pas de temps

➔erreur proportionnelle à dt² au lieu de dt
  - Plus stable mais reste des instabilités




**D'autres méthodes d'intégrations existent**



## Schéma d'intégration

- Verlet

$$P_{t+\Delta t} = 2P_t - P_{t-\Delta t} + \Delta t^2 \times A_t$$

**D'autres méthodes d'intégrations existent**

## Position Based Dynamics

Développeur de NVIDIA Physics
http://matthias-mueller-fischer.ch/

- Mise à jour de V :

$$mA_t = F$$
$$m\frac{dV}{dt} = F$$
$$m\frac{V_t - V_{t-\Delta t}}{\Delta t} = F$$
$$V_t = V_{t-\Delta t} + \Delta t\,\frac{F}{m}$$

- Puis mise à jour de P :

$$\frac{dP}{dt} = V_t$$
$$\frac{P_t - P_{t-\Delta t}}{\Delta t} = V_t$$
$$P_t = P_{t-\Delta t} + \Delta t \times V_t$$

*Variant : leap-frog, Stoermer-Verlet*



## Position Based Dynamics

http://matthias-mueller-fischer.ch/

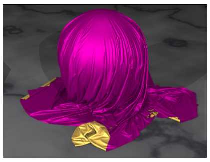
| | |
|---|---|
| $m_i$ | mass |
| $\mathbf{x}_i$ | position |
| $\mathbf{v}_i$ | velocity |

**Algorithm 1** Position-based dynamics

**for all** vertices $i$ **do**
  initialize $\mathbf{x}_i = \mathbf{x}_i^0, \mathbf{v}_i = \mathbf{v}_i^0, w_i = 1/m_i$
**end for**
**loop**
  **for all** vertices $i$ **do** $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t\, w_i \mathbf{f}_{ext}(\mathbf{x}_i)$
  **for all** vertices $i$ **do** $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t\, \mathbf{v}_i$
  **for all** vertices $i$ **do** genCollConstraints($\mathbf{x}_i \rightarrow \mathbf{p}_i$)
  **loop** solverIteration **times**
    projectConstraints($C_1, \ldots, C_{M+M_{Coll}}, \mathbf{p}_1, \ldots, \mathbf{p}_N$)
  **end loop**
  **for all** vertices $i$ **do**
    $\mathbf{v}_i \leftarrow (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$
    $\mathbf{x}_i \leftarrow \mathbf{p}_i$
  **end for**
  velocityUpdate($\mathbf{v}_1, \ldots, \mathbf{v}_N$)
**end loop**



## Contraintes, collisions



- Exemple : sol en Y=0 y<0 ?

➔ Différentes solutions
  - Appliquer la force $F_{sol}$ de réaction du sol proportionnelle à la distance sous le sol
  - Autorise la pénétration sous le sol mais la force va corriger le tissus au pas de temps suivant
- OU
  - …

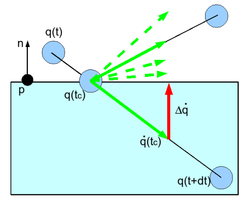
Avec ∑F=ma on a équivalence entre F et déplacement



## Collisions avec retour arrière

- Détection d'une collision puis
  - Revenir en arrière au moment de la collision
  - Puis calcul de la mise à jour de V : $\Delta V = -(V.n)n$
  - Applique un coefficient r de rebond : V += (1+r). ΔV

\- Problème : assez lourd de revenir en arrière quand les 2 objets sont en mouvements

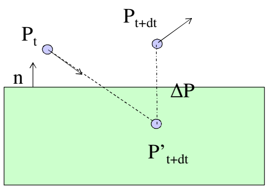



Sur la figure P=q et V= $\dot{q}$



## Collisions

- Détection d'une collision puis
  - Mise à jour de P : ΔP = 2 x vecteur entre P' et le sol (symétrique de P' par rapport au sol)
  - Applique un coefficient r de rebond : P+=(1+r) ΔP
  - Mise à jour de V : symétrique par rapport au sol

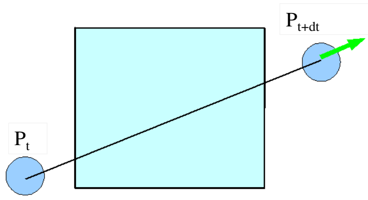



\+ Résolutions simultanées des collisions de tous les objets en même temps



## Collisions

- Cas limite de détection de collision avec pas de temps discret



## Solide rigide

- Idem point
  - Physique du point : ∑F=ma
- + orientation (rotation)
  - ∑L=I.ω
  - Somme des moments des forces extérieurs = moment d'inertie du solide * Accélération angulaire

Remarque : Un mouvement au sens le plus général peut être considéré à chaque instant comme la superposition d'une translation et d'une rotation autour d'un axe.(par exemple le mouvement d'une bille sur un plan incliné) Pour résoudre les équations du mouvement, les 2 équations si dessus sont nécessaires

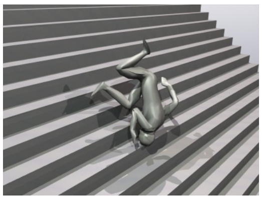


## Fin de parenthèse : physique)



## Ragdoll = poupée désarticulée

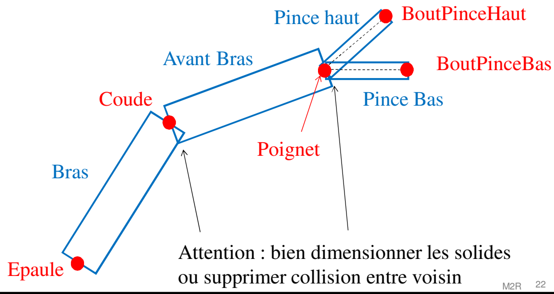
- Poupée désarticulée
  - Corps = ensemble de membres + articulation
  - Membre = solide rigide (cube, capsule, etc.)
  - Articulation = contrainte de liaison entre les membres
- Vidéo
- Ou démo Bullet
  - http://bulletphysics.org/





## Ragdoll = poupée désarticulée

- Articulation = contraintes
- Membre = solide rigide animé






## Physique : Newton + contraintes

- Un cycle de calcul physique =
  - Equation physique sur chaque partie du corps
    - Newton : ∑F=ma et ∑L=I.ω
  - Résolution des contraintes
    - Connexion des articulations
    - ➔pour ragdoll globalement 2 méthodes :

Featherstone, R. (1987). *Robot Dynamics Algorithms.* Kluwer. ISBN 0-89838-230-0.

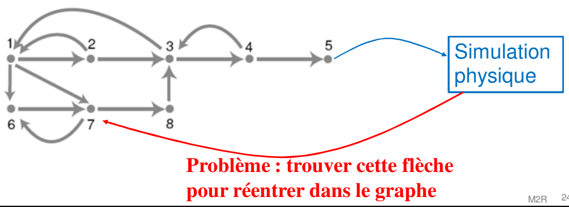
ou

D. Baraff. Linear-time dynamics using Lagrange multipliers. *SIGGRAPH* 1996



## MoCap/Physique : graphe d'animation

- Graphe d'animation + physique
  - Comporte des nœuds de sortie vers la physique
- Utilisable seulement dans certains cas
  - Chute, coups, …



## MoCap/Physique : graphe d'animation

- Graphe d'animation + physique
  - Problème : rentrer dans le graphe après la physique
    ➔ Anticiper quelques frames de physique et chercher les similitudes de positions
    ➔ Ajouter des forces « virtuelles » sur les articulations pour les diriger vers une position du graphe
    ➔ Demande un graphe de MoCap bien remplit avec des séquences pour se relever, rouler, etc.

VIDEO



## MoCap/Physique : graphe d'animation

- **Forces « virtuelles »**
  - Amener chaque articulation vers l'angle désiré
  - Proportional-derivative (PD) control

$$\tau = \underbrace{k_p(\theta_d - \theta)}_{\text{Respond to changes.}} - \underbrace{k_d\dot{\theta}}_{\text{Damp}}$$



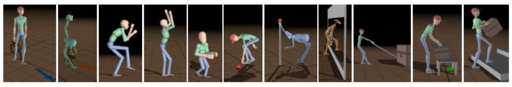
## Encore plus loin : tout physique

- Objectifs
  - Animations plus réalistes/réactives : poids, fatigue, etc.
  - Editer les effets physiques : changer gravité, etc.
- Principe
  - On part d'un ragdoll et on essaie de lui donner un contrôle moteur = un « cerveau » dédié à l'animation
  - Un graphe d'animation peu jouer le rôle de « mémoire » de mouvements
- Problèmes
  - Contrôle de l'équilibre (*balance control strategy*)
  - Combiner le mouvement entre la mocap et le réactif
  - …



## Encore plus loin : tout physique

- Domaine de recherche
  - SIMBICON (SIMple BIped CONtroler) 2007
    - A bien relancé l'idée en recherche CG

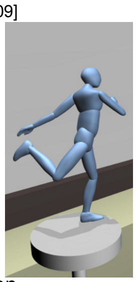


- Idée très présente en robotique
  - incompatibilité entre mocap humaine et robot
  - Avec des problèmes supplémentaires
    - Contrôle/Réactivité des moteurs



## Stratégie : contrôle de l'équilibre

- Stratégie qui semble marcher [MZS09,JYL09]
  - Optimisation + machine à état
  - → Problème complexe à l'état de recherche

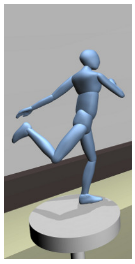


- Optimisation : Fonction = stabilité
  - Projection du centre de gravité sur le sol
  - Comparer à la position des pieds
  - → Fournir une fonction indicateur de stabilité
  - Dépend des angles entre chaque articulation
    ➔ Non linéaire (demande des méthodes d'optimisation adaptée)



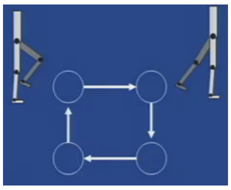
## Stratégie : contrôle de l'équilibre

- Stratégie qui semble marcher [MZS09,JYL09]
  - Optimisation + machine à état
  - → Problème complexe à l'état de recherche
- Machine à état
  - 2 pieds au sol
    - Optimisation de la position des bras et du buste pour maintenir l'équilibre
  - Lève un pied
    - Optimisation des bras, du buste et de la jambe
  - Etc.
- **VIDEO**

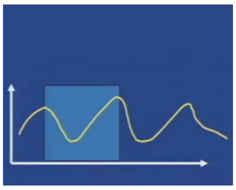


## Marche : Machine à états finis

- Marche cyclique



Automate Machine à états finis
*(FSM : Finite State Machines)*

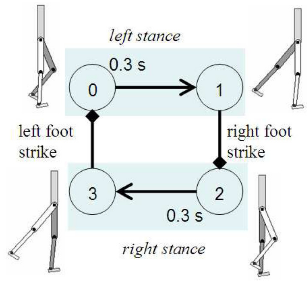


Motion capture



## Marche : machine à états finis

- 4 états :



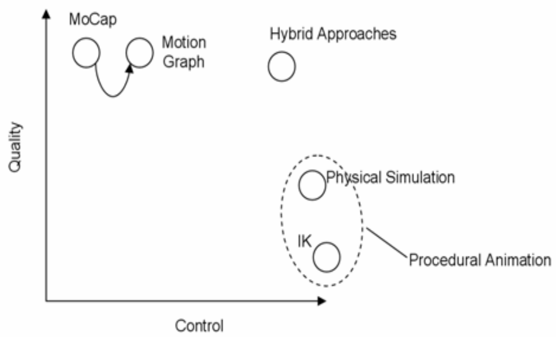

state 0 → state1 : after 0.3s

state 1 → state 2 : after swing foot contact

(SIMBICON) … VIDEO



## Conclusion : animation de perso





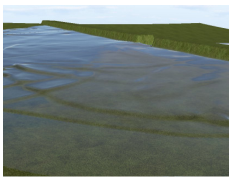

## Animation physique d'une surface d'eau

- Equation de Saint-Venant ou Shallow Water Equation
- TP : OpenGL + OpenCL

## Equation de Saint-Venant

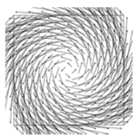
Equations de Saint-Venant ou Shallow Water equations

- Dérivée des eq. de Navier-Stokes
- Real Time Physics, Siggraph 2008 class.
- décrit les écoulements naturels : surface libre et eau peu profonde
- Grille 2D Approche Eulerienne (!= particule, approche Lagrangienne)



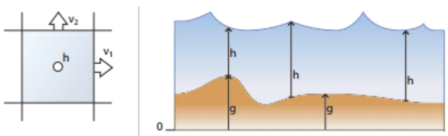


## Equation de Saint-Venant



Champ de vecteur v

- Grille 2D
- h hauteur du fluide
- g hauteur du sol
- n hauteur du fluide sur le sol : n = h–g.
- $v(v_1,v_2)$ vitesse du fluide dans le plan horizontal



## Equation de Saint-Venant : intuitif

- Equations de Saint-Venant (Shallow Water)

$$\frac{\partial \eta}{\partial t} + (\nabla \eta)\mathbf{v} = -\eta \nabla \cdot \mathbf{v} \qquad (1) \text{ avec } \nabla \cdot \boldsymbol{A} \equiv \operatorname{div} \boldsymbol{A} = \frac{\partial A_x}{\partial x} + \frac{\partial A_y}{\partial y} + \frac{\partial A_z}{\partial z}$$

$$\frac{\partial \mathbf{v}}{\partial t} + (\nabla \mathbf{v})\mathbf{v} = a_n \nabla h\,, \qquad (2)$$

- $a_n$ : accélération verticale du fluide (gravité)=9.81
- (1) s'occupe de la variation de quantité d'eau
- (2) s'occupe de la variation de la vitesse
- Partie gauche : calculer par advection

L'advection correspond au transport d'une quantité (scalaire ou vectorielle) par un champ vectoriel.



## Equation de Saint-Venant

- Equation de Saint-Venant (Shallow Water)

$$\begin{aligned} \partial \eta/\partial t + (\nabla \eta)\mathbf{v} &= -\eta \nabla \cdot \mathbf{v} && (1)\\ \partial v_1/\partial t + (\nabla v_1)\mathbf{v} &= a_n \nabla h && (2)\\ \partial v_2/\partial t + (\nabla v_2)\mathbf{v} &= a_n \nabla h\,. && (3) \end{aligned}$$

- Partie gauche : calculer par advection

$$\begin{array}{l} \textit{Advect}(s, \mathbf{v})\\ \quad \textbf{for } j = 1 \textbf{ to } n_2 - 1\\ \quad\quad \textbf{for } i = 1 \textbf{ to } n_1 - 1\\ \quad\quad\quad \mathbf{x} = (i \cdot \Delta x, j \cdot \Delta x)\\ \quad\quad\quad \mathbf{x}' = \mathbf{x} - \Delta t \cdot v(\mathbf{x})\\ \quad\quad\quad s'(i,j) = \text{interpolate}(s, \mathbf{x}')\\ \quad\quad \textbf{endfor}\\ \quad \textbf{endfor}\\ \quad \textbf{return}(s') \end{array}$$

L'advection correspond au transport d'une quantité (scalaire ou vectorielle) par un champ vectoriel.



## Equation de Saint-Venant : intuitif

- Equations de Saint-Venant (Shallow Water)

$$\frac{\partial \eta}{\partial t} + (\nabla \eta)\mathbf{v} = -\eta \nabla \cdot \mathbf{v} \qquad (1)$$

$$\frac{\partial \mathbf{v}}{\partial t} + (\nabla \mathbf{v})\mathbf{v} = a_n \nabla h\,, \qquad (2)$$

- (1) correspond à la variation de quantité d'eau
  - Variation de la quantité d'eau = dn/dt = -n∇.v = en fonction du champ de vitesse, on fait le bilan des arrivées des départs en eau
  - Si plus d'eau part, que d'eau arrive (∇.v>0), ca descend.
  - Et inversement.



## Equation de Saint-Venant

- Equation de Saint-Venant (Shallow Water)

$$\begin{aligned} \partial \eta/\partial t + (\nabla \eta)\mathbf{v} &= -\eta \nabla \cdot \mathbf{v} && (1)\\ \partial v_1/\partial t + (\nabla v_1)\mathbf{v} &= a_n \nabla h && (2)\\ \partial v_2/\partial t + (\nabla v_2)\mathbf{v} &= a_n \nabla h\,. && (3) \end{aligned}$$

- Partie gauche : calculer par advection
- (1) donne variation de n donc n' = n – n ∇.v

$$\begin{array}{l} \textit{Update-height}(\eta, \mathbf{v})\\ \quad \textbf{for } j = 1 \textbf{ to } n_2 - 1\\ \quad\quad \textbf{for } i = 1 \textbf{ to } n_1 - 1\\ \quad\quad\quad \eta(i,j) -= \eta(i,j) \cdot \left( \frac{(v_1(i+1,j) - v_1(i,j))}{\Delta x} + \frac{(v_2(i,j+1) - v_2(i,j))}{\Delta x} \right) \Delta t\\ \quad\quad \textbf{endfor}\\ \quad \textbf{endfor}\\ \quad \textbf{return}(\eta') \end{array}$$



## Equation de Saint-Venant : intuitif

- Equations de Saint-Venant (Shallow Water)

$$\frac{\partial \eta}{\partial t} + (\nabla \eta)\mathbf{v} = -\eta \nabla \cdot \mathbf{v} \qquad (1)$$

$$\frac{\partial \mathbf{v}}{\partial t} + (\nabla \mathbf{v})\mathbf{v} = a_n \nabla h\,, \qquad (2)$$

- (2) correspond à : m.a = m.dv/dt = ∑F
  - Ici F=a∇h
  - le fluide subit une force dans la direction : de plus d'eau vers moins d'eau



## Equation de Saint-Venant

- Equation de Saint-Venant (Shallow Water)

$$\begin{aligned} \partial \eta/\partial t + (\nabla \eta)\mathbf{v} &= -\eta \nabla \cdot \mathbf{v} && (1)\\ \partial v_1/\partial t + (\nabla v_1)\mathbf{v} &= a_n \nabla h && (2)\\ \partial v_2/\partial t + (\nabla v_2)\mathbf{v} &= a_n \nabla h\,. && (3) \end{aligned}$$

- Partie gauche : calculer par advection
- (2)(3) donnent variation de v donc
  - v1 = v1 + $a_n \nabla h_1$ avec $\nabla h_1$ = dh/dx
  - v2 = v2 + $a_n \nabla h_2$ avec $\nabla h_2$ = dh/dy

$$\begin{array}{l} \textit{Update-velocities}(h, v_1, v_2, a)\\ \quad \textbf{for } j = 1 \textbf{ to } n_2 - 1\\ \quad\quad \textbf{for } i = 2 \textbf{ to } n_1 - 1\\ \quad\quad\quad v_1(i,j) += a\left(\frac{h(i-1,j) - h(i,j)}{\Delta x}\right)\Delta t\\ \quad\quad \textbf{endfor}\\ \quad \textbf{endfor}\\ \quad \textbf{for } j = 2 \textbf{ to } n_2 - 1\\ \quad\quad \textbf{for } i = 1 \textbf{ to } n_1 - 1\\ \quad\quad\quad v_2(i,j) += a\left(\frac{h(i,j-1) - h(i,j)}{\Delta x}\right)\Delta t\\ \quad\quad \textbf{endfor}\\ \quad \textbf{endfor} \end{array}$$

## Saint-Venant : intégration

- Shallow-water-step(h,v,g)
  - (1) n = Advect(n,v)
  - (2) v1 = Advect(v1,v)
  - (3) v2 = Advect(v2,v)
  - (4) n' = Update-height(n,v)
  - (5) h = n'+g
  - (6) Update-velocities(h,v1,v2)
  - (7) n=n'



## OpenCL (wikipedia)

- **OpenCL** (**Open** **C**omputing **L**anguage) est la combinaison d'une API et d'un langage de programmation dérivé du C, proposé comme un standard ouvert par le Khronos Group.
- OpenCL est conçu pour programmer des systèmes parallèles hétérogènes comprenant par exemple à la fois un CPU multi-cœur et un GPU.
- OpenCL propose donc un modèle de programmation se situant à l'intersection naissante entre le monde des CPU et des GPU, les premiers étant de plus en plus parallèles, les seconds étant de plus en plus programmables.



## OpenCL

- OpenCL can accelerate code by a factor 10 or more
- OpenCL is an open standard
- OpenCL can help save power
- OpenCL can save you hardware cost
- OpenCL adoption is ramping up rapidly
- OpenCL may be used as the basis for generating custom hardware
- The OpenCL C99 language is based on C
- OpenCL can be used from a variety of host languages
- It is easy to start with OpenCL
- OpenCL is platform independent

http://www.amdahlsoftware.com/ten-reasons-why-we-love-opencl-and-why-you-might-too/



## OpenGL / OpenCL : le TP

- Le code de départ
  - Carte de hauteur dans une texture 2D (GL)
  - Affichage GL avec un vertex shader
    - Vertex.glsl
  - kernel OpenCL modifie la texture
    - Texture 2D = vu comme une 'image2d'
    - CLWater.cl
    - Kernel = fonction appelée sur chaque case du tableau



## OpenCL : un kernel

```
__kernel void shallowWaterInit( const int dimD,
                                __global __write_only image2d_t DD0 )
{
        const int x = get_global_id(0);
        const int y = get_global_id(1);

        if (x>=dimD) return; if (y>=dimD) return;
        const sampler_t sampler =
CLK_NORMALIZED_COORDS_FALSE | CLK_ADDRESS_CLAMP |
CLK_FILTER_NEAREST;

        float4 pixel={0,0,0,0};
        write_imagef( DD0, (int2)(x,y), pixel);
}
```



## OpenCL : l'appel

```
clSetKernelArg( m_kernelShallowWater, 0, sizeof(Dim), (void*)&Dim);
clSetKernelArg( m_kernelShallowWater, 1, sizeof(D0), (void*)&D0);

const size_t localWorkSize[] = { LocalWorkSize, LocalWorkSize };
const size_t globalWorkSize[] = { Dim, Dim };

cl_uint workDim = 2;
clEnqueueNDRangeKernel ( m_queue, m_kernelShallowWaterInit,
                         workDim, NULL,
                         globalWorkSize, localWorkSize,
                         0, NULL, NULL);
```

TP animation physique
de personnage
avec BulletPhysics

## TP

- Combiner MoCap et Ragdoll
  - Utiliser Bullet pour la physique
- Regardez le fichier Ragdoll.h/.cpp
  - Ce ragdoll doit avoir la même configuration que le squelette de la MoCap
  - Pour l'instant, c'est un squelette avec 2 membres (bras, avant-bras) qui sont entrés en dur.



## TP : Lib BulletPhysics

- Le monde physique gérés par la lib
  - btDynamicsWorld* m_dynamicsWorld;

➔Les objets physiques doivent y être ajoutés

- Dans le TP, il y a une class CAPhysics qui s'occupe du monde physique de Bullet avec
  - computePhysics() : calcul la physique depuis le dernier appel
  - renderPhysics() : affiche les objets physiques, appuyer sur 'P' dans le viewer pour les afficher
  - createRigidBody(…) : ajoute un objet rigide dans le monde physique et renvoie un pointeur dessus



## TP : Lib BulletPhysics

- Ragdoll = ensemble de
  - vector<btRigidBody*> m_bodies;
    - Solides rigides
  - vector<btCollisionShape*> m_shapes;
    - Les formes pour les collisions (optionnel)
  - vector<btTypedConstraint* > m_jointsConstraint;
    - Les articulations



## TP : Lib BulletPhysics

- Bullet gère les transformations
  - btTransform transform;
    - Rotation + translation
  - Construit avec un quaternion + un vecteur
    - btTransform t( q, v)
  - Construit avec une matrice 3x3 + vecteur*

…

  - Toutes les fonctions dont vous avez besoin sont disponibles …



## TP : Lib BulletPhysics

- Créer un corps solide btRigidBody
  - Utiliser la fonction de CAPhysics

```
btRigidBody* CAPhysics::createRigidBody(
                float mass,
                const btTransform& startTransform,
                btCollisionShape* shapeForCollision)
```

- Créer une forme pour les collisions

```
new btCapsuleShape( rayon, hauteur );
```

## TP : Lib BulletPhysics

- Créer une articulation

coneC = new btConeTwistConstraint(A, B, localA, localB);

  - A et B sont des btRigidBody
  - localA et localB sont des btTransform relatif à au repère local du btRigidBody
- Articulation avec des contraintes en forme de Cone

- Il existe d'autres types d'articulation
  - btHingeConstraint : mouvement dans le plan type doude
  - …

- Ne pas oublier d'ajouter les articulations au monde physique
  - m_physics.getDynamicsWorld()->addConstraint( coneC, true);
  - true pour ne pas calculer de colision entre 2 btRigidBody relié par l'articulation



## TP : Lib BulletPhysics

- Regardez les 2 exemples dans CARagdoll.h