دانشکده مهندسی کامپیوتر

پیاده‌سازی یک موتور فیزیک با زبان جاوا اسکریپت جهت توسعه بازی‌های تحت وب در قالب HTML5

پایان‌نامه برای دریافت درجه کارشناسی

کامران نوبهار

۸۶۵۲۲۰۹۵

استاد راهنما:

دکتر بهروز مینایی

دی ماه ۱۳۹۱

به نام خدا

چکیده

در این گزارش به بررسی جدیدترین امکانات معرفی شده در HTML5 و همچنین نحوه پیاده‌سازی یک موتور فیزیک جهت استفاده از یکی از مهم‌ترین عناصر جدید HTML یعنی canvas می‌پردازیم.

در ابتدا با معرفی مهم‌ترین تکنولوژی‌های وب، تاریخچه پیدایش، رشد و مقبولیت آن‌ها به بررسی بسترهای ظهور عناصر جدید HTML5 پرداخته و جایگاه حال و آینده این عناصر و همچنین گزینه‌های پیشینشان را مطالعه‌ای کلی می‌کنیم. همچنین در مقدمه این مسئله مطرح می‌شود که چرا canvas به عنوان یکی از عناصر تازه در HTML به عنوان رکن اصلی این پروژه برای هدف قرار گرفتن کل پیاده‌سازی انتخاب شده است. به طور کلی می‌توان گفت که canvas پس از ظهور و حکم فرمایی تعداد متنوعی از ساختارهای نرم افزاری برای ارائه محتوای چندرسانه‌ای می‌تواند به عنوان جایگزینی جامع و استاندارد استفاده شود و در این پروژه نشان داده می‌شود که چطور یک موتور فیزیک برای ایجاد بازی‌هایی در این ساختار می‌تواند ساخته شود و کار کند.

Canvas اما یک محیط گرافیکی برداری است که در درون یک سند HTML5 جای می‌گیرد و برای دسترسی به و دستکاری در آن می‌بایست از درخت DOM سند استفاده کرده و با JavaScript با کار با آن بپردازیم. رابط کاربری معرفی شده با canvas یک context دو بعدی در اختیار می‌گذارد که با رابط کاربری آن می‌توانیم انواع اشکال برداری را در Canvas رسم نماییم. این رابط کاربری شامل انواع توابع مختلف و متنوع برای ترسیم هر شکل، تصویر و یا حتی دستکاری‌های پیکسلی می‌باشد. در فصل یک به معرفی این توابع و قابلیت‌ها و همچنین دیگر مطالب فنی مرتبط می‌پردازیم.

Canvas از آن جایی که عنصر بسیار جدیدی در وب است، ابزارها و کتابخانه‌های کافی برای کار و استفاده با آن هنوز وجود ندارد، یکی از این کمبودها یک موتور فیزیک جامع برای ساخت و ایجاد بازی‌های

دوبعدی است. موتور فیزیک در واقع یک قطعه کد است که وظیفه محاسبات مربوط به شبیه‌سازی فیزیکی اشیا درون یک بازی با اشیا ما به ازا آن‌ها در دنیای واقعی را دارد. ضرورت ایجاد این موتور فیزیک به طور جدا از ضرورت کلی وجود آن نیست و چنان که بعدا در فصل دو بیشتر توضیح خواهیم داد صرفه جویی در کد و بالا بردن کیفیت بازی‌های کامپیوتر دو دلیل عمده ضرورت ایجاد این موتور است.

در این پروژه ابتدا یک موتور فیزیک ذره‌ای و سپس با گسترش آن یک موتور فیزیک تجمیع ذرات را پیاده‌سازی می‌کنیم که در فصل سه و چهار به معرفی ویژگی‌های آن و همچنین مبانی اولیه ریاضی و فیزیک مربوط به آن می‌پردازیم و آن چه را که در موتور قصد شبیه‌سازی داریم مطالعه خواهیم کرد.

برای گسترش موتور فیزیک سپس بحث نیروها و مولدهای نیرو پیش کشیده می‌شود و نیروهای مختلف نیز باید در موتور گنجانده شود که در فصل بعد به آن پرداخته می‌شود. همچنین مهمترین موجودیت موتور تجمیع ذرات که محدودیت‌ها هستند نیز در موتور ضروری می‌باشند. این موجودیت‌ها نیز در فصل هفت و هشت بررسی و نحوه پیاده‌سازی آن‌ها بیان می‌شود.

در انتهای این گزارش خواهیم دید که چطور در این پروژه یک موتور فیزیک تجمیع ذرات با زبان جاوا اسکریپت برای استفاده در محیط canvas پیاده‌سازی شده است.

فهرست مطالب

فهرست شکل‌ها

مقدمه

گستره جهانی وب به عنوان مهم‌ترین بستر انتقال اطلاعات و داده و همچنین ارتباطات دیگر در شبکه اینترنت شناخته می‌شود، آن چنان که انواع ارتباطات دیجیتال چه از نوع انتقال صرف اطلاعات و یا ارتباطات تعاملی را پشتیبانی می‌کند. مرورگران وب مهمترین و در واقع رکن اصلی در این ارتباط می‌باشند و بستر نمایش و تعامل با انواع داده‌های وب در اینترنت را فراهم می‌آورند. در سال ۱۹۸۰ استانداردی برای انتقال اسناد در شبکه‌های کامپیوتری مرکز CERN تدوین شد تا انتقال اطلاعات بتواند در یک قالب مشخص و کارآمد انجام بگیرد. این استاندارد که HTML نام گرفت بعدا به مهمترین زبان تدوین اطلاعات در وب بدل شد، و می‌توان گفت که هم اکنون اکثر داده‌ها در وب در شبکه اینترنت با این زبان انتقال می‌یابند.

HTML که در واقع زبان نشانه‌گذاری ابرمتنی است، در ابتدا مهمترین هدفش تدوین ساختاری بود که کاربر بتواند به راحتی در میان اسناد وب گردش کند و از یک به متن به متن دیگری رجوع کند، اما کم کم با پیشرفت تکنولوژی‌های انتقال اطلاعات و به تبع آن بالارفتن هر چه بیشتر پهنای باند اینترنت امکان این به وجود آمد تا داده‌های دیگری را نیز بتوان از طریق وب جابجا کرد، که از آن جمله می‌توان به عکس‌ها، فیلم‌ها، فایل‌های صوتی و غیره اشاره کرد. در نتیجه لازم بود تا زبان HTML بتواند این نوع داده‌ها را نیز در خود جای دهد، تا بتوان آن‌ها در کامپیوترهای کاربری دریافت و مشاهده کرد. نوع و شکل داده‌های چند رسانه‌ای این گونه کم کم زیاد شده و با حساب انواع فرمت‌های هر رسانه اعم از فیلم، موسیقی و عکس تلاش‌های زیادی لازم بود تا با استاندارد کردن آن‌ها در قالب HTML و احیانا خارج از آن در بستر وب امکان دریافت و ارسالشان به طور گسترده به وجود آید. در این میان مرورگران وب نقشی حیاتی داشتند چرا که می‌توان گفت تنها ابزار دریافت، نمایش و احیانا تعامل با وب در محیط کاربر بوده اند، و وظیفه پشتیبانی از انواع استانداردها و فرمت‌های به وجود آمده و در حال به وجود آمدن برای عرضه به کاربر را بر عهده داشتند.

در میان انبوه شکل‌های مختلف داده‌های چند رسانه‌ای بعضی بیشتر مورد استقبال قرار می‌گرفتند و بعضی کمتر همچنین کنسرسیوم گستره جهانی وب که مسئولیت استانداردسازی و پشتیبانی از استانداردهای وب را دارد، به بعضی انواع داده‌ها بیشتر توجه کرده و به بعضی کمتر. برای مثال فرمت‌های ویدئویی و صوتی در ابتدای عرضه شان در وب و تا چند سال پیش تنها از طریق برنامه‌های جانبی دیگری قابل پخش بودند که باید در محیط کاربر نصب می‌شدند. روش انتقال این داده‌ها در نتیجه به این شکل می‌بود که در صفحات فرستاده شده از طریق سرور وب به کاربر که با HTML استاندارد شده است، در درون سند، آدرس اینترنتی فایل صوتی یا تصویری داده می‌شد تا از این طریق مرورگر بتواند با بارگذاری آن از آن آدرس فایل را دریافت کند، پس از دریافت فایل مرورگر با توجه به فرمت فایل افزونه مناسب را که قابلیت پخش و ارائه فایل را داشته و قبلا می بایست نصب شده می‌بود و به مرورگر الصاق می‌شد فراخوانی کند، تا افزونه که معمولا با یک برنامه اصلی و جدا از مرورگر و نصب شده در محیط کاریر در ارتباط است فایل را در خود محیط مرورگر و در محلی از سند که توسط زبان HTML مشخص شده است پخش کند و نمایش دهد. برای مثال اگر فرمت ویدئویی و صوتی WMA را در نظر بگیریم، وقتی کاربری به صفحه‌ای از وب رجوع می‌کند که حاوی فایلی از این فرمت است، مرورگر کاربر ابتدا چک می‌کند که آیا افزونه‌ای را در اختیار دارد که بتواند این فایل را با کمک آن نشان دهد، و اگر نه پیغام خطایی را به او نشان می‌دهد که افزونه مناسب فایل WMA باید نصب شود. فایل WMA فرمت اختصاصی برنامه Windows Media Player می‌باشد و در محیط ویندوز تنها این برنامه قادر به پخش آن می باشد، در نتیجه لازم است که اولا خود برنامه در محیط نصب شده و دوما افزونه Windows Media Player که برای مرورگر خاص مثلا Internet Explorer ساخته شده نیز نصب و به IE اضافه شود. به این صورت IE با فراخوانی آن افزونه و آن افزونه با فراخوان برنامه اصلی (WMP) فایل را در محل مناسب خود پخش می‌کند.

در این میان دو تکنولوژی مهم دیگر در محیط وب رشد کردند. اول تکنولوژی اسکریپت نویسی در اسناد HTML بود که از این طریق امکان پویا کردن و تعاملی کردن این صفحات به وجود آمد. مهمترین زبان

اسکریپت نویسی در این میان JavaScript بوده که امکانات بسیاری را در این زمینه به عرضه گذاشت. با قابلیت های این زبان این امکان به وجود آمد که به صورتی برنامه ریزی شده بتوان به عناصر درونی اسناد HTML دسترسی داشت و آنها را تغییر داد، بدین صورت اولین امکان تعامل در طرف کاربر با سند وب به وجود آمده و سند وب می‌توانست به عنوان یک برنامه کامپوتری ساده عمل کند و برای مثال از کاربر ورودی گرفته و پیغام مناسبی را چاپ کند. لازم به ذکر است که مانند تمام عناصر دیگر وب وظیفه اجرای اسکریپت‌ها بر عهده مرورگر بوده اما بر خلاف عناصری چون Flash افزونه جداگانه‌ای برای اجرای آن لازم نمی‌بود چرا که اسکریپت ها زبانی‌هایی هستند که خود مرورگر می‌تواند تفسیر آنها را به عده بگیرد.

اما تکنولوژی مهم دوم که ابتدا اصلا در خارج از وب و اینترنت به وجود آمد، بستر Flash بود. Flash یک بستر و فرمت ارائه شده توسط شرکت Macromedia بود که برای ایجاد و نمایش انواع داده‌های چند رسانه ای می‌توانست به کار گرفته شود، تکنولوژی کشیدن اشکال بر پایه بردارها آن را تبدیل به قدرتمندترین فرمت شناخته شده در زمان خود برای ارائه داده چند رسانه‌ای کرد. در Flash همچنین بعدا امکان ایجاد انیمیشن‌های برداری و کم کم با اضافه شدن قابلیت‌های اسکریپت نویسی در درونش امکان ایجاد برنامه‌های تعاملی از جمله بازی‌های ویدئویی به وجود آمد. این امکانات که در ابتدا در محیط‌های خارج از وب ارائه شده بودند در مقایسه با دیگر برنامه‌هایی که امکان ایجاد بازی و انیمیشن را می‌دادند برتری به نسبه‌ای به Flash دادند چنان که شرکت Macromedia تصمیم گرفت فایل‌های با فرمت Flash را در اینترنت و در وب تیز ارائه کند. این ارائه منجر به یکی از بزرگترین تحولات در وب شد چنان که افزونه‌ای که برای نمایش Flash به کار می‌رود تبدیل به مهمترین و با بیشترین تعداد استفاده از میان افزونه‌های مرورگران وب شد. گستردگی Flash به گونه‌ای بود و می‌توان گفت که هم اکنون نیز هست که بسیاری از وب سایت‌های اینترنتی تمام محتوای خود را از طریق آن ارائه می‌کنند و در اکثر وبسایت‌های دیگر رد پای آن را می‌توان یافت، از جمله استفاده گسترده در تبلیغات محیطی وبسایت‌ها و یا ارائه داده‌هایی که نیاز به انیمیشن دارند. راحتی استفاده و همچنین محیط زیبا و مورد

پسند Flash و همچنین نبود رقیب جدی برای آن در محیط وب از جمله مهمترین عوامل موفقیت این تکنوژی در وب می‌باشد. Flash حتی با پیشرفتی که خود داشته توانست نحوه ارائه فایل‌های ویدئویی و صوتی را نیز متحول کند و پلیرهایی برای اجرای آن‌ها ارائه دهد. یکی دیگر از کاربردهای وسیعی که Flash در محیط وب به وجود آورد امکان ایجاد بازی‌های دو بعدی و حتی سه بعدی در این محیط بود، به طوری که حالا می‌شد در یک سند HTML در محیط وب بتوان بازی کرد. هر چند این گونه بازی‌ها در مقایسه با بازی‌های ویدئویی که در محیط کامپیوتری و با کمک کارت‌های گرافیک قدرتمند اجرا می‌شوند ساده به نظر بیایند اما واقعیت این است که گستردگی و استفاده و محبوبیت بسیاری داشته‌اند. فراموش نباید کرد که Flash نیز مانند دیگر برنامه‌های خارج از وب برای اجرا شدن در محیط مرورگر نیاز به یک افزونه برای مرورگر و همچنین یک برنامه اصلی در خارج از مرورگر دارد و به همان طریق پیشتر گفته شده اجرا می‌شود.

نمایش فایل‌های چند رسانه‌ای به نوع‌هایی که توضیح داده شد در وب تا مدتی متدوال بود، اما کاربران اینترنتی کم کم متوجه شدند که بسیار روش ناکارآمدی است و مشکلات عدیده‌ای را به همراه دارد. از جمله این مشکلات می‌توان به دشواری پشتیبانی محیط کاربری از تمام فرمت‌های موجود با توجه به تنوع روز افزون انواع فایل‌های چند رسانه‌ای اشاره کرد. به این صورت که کاربری که بخواهد انواع فایلها را در کامپیوتر خود از وب مشاهده کند می‌بایست انواع افزونه‌ها و برنامه‌های مختلف را نصب کند و به محض برخورد با یک فرمت جدید مجبور خواهد شد تا یک افزونه جدید بارگذاری و نصب کند. مشکل دیگر خطرات عدیده امنیتی بود که از این طریق کاربران را تهدید می‌کرد. افزونه‌های ارائه شده برای مرورگرهای مختلف برای فرمت‌های مختلف توسط شرکت‌های گوناگونی ایجاد و عرضه می‌شدند و با توجه به این که کنترلی بر روی محتوای فایل‌های غیر از اسناد HTML نمی‌تواند وجود داشته باشد، آسیب پذیری‌های امنیتی زیادی می‌تواند در افزونه‌ها امنیت کاربران را به خطر اندازد. مرورگر هیچگونه کنترلی بر روی اجرای افزونه‌ها نداشته و ارتباط با یک برنامه دیگر در محیط کاربری به شدت خطرناک می‌باشد. فایل‌های صوتی و تصویری می‌توانند بالقوه فایل‌های اجرای مخربی باشند که

به راحتی از کنترل‌های امنیتی مرورگر عبور کرده و از حفره‌های امنیتی افزونه‌ها استفاده کنند. بیشترین تعداد مشکلات امنیتی سال‌های اخیر در محیط وب در رابطه با افزونه Flash بوده که هشدارهای جدی را در رابطه با این افزونه در پی داشته است. در واقع متن بسته بودن افزونه‌ها و برنامه‌هایی چون Flash همواره نگرانی کسانی که در پشتیبانی گستره جهانی وب نقش داشته اند را برانگیخته است. چنان که کنسریوم جهانی وب نیز همواره بر این نکته تاکید داشته است که روحیه جاری در محیط اینترنت بر متن باز بودن اطلاعات انتقال یافته و فایل های اجرایی متکی است.

این موضوع شاید مهمترین دلیل تلاش‌های بعدی این کنسرسیوم و افراد دیگری با همین دغدغه برای تدوین استانداردهای نو و یک پارچه برای نمایش و اجرای فایل‌های چند رسانه‌ای در محیط وب بوده است. بدین صورت تلاش شد تا سایه افزونه‌های گوناگون با مشکلات جدی از سر مرورگرها کم شود و خود آن‌ها مسئول اجرا و نمایش فرمت‌های گوناگون شوند. بدین صورت تمام تلاش‌ها برای یک پارچه‌سازی و استانداردسازی این فرمت‌ها در تدوین پنجمین نسخه HTML جمع شد و در آن به بار نشست. HTML5 که تغییرات بسیاری نسبت به نسخه پیشین خود کرد، بسیاری از مشکلات قبلی را حل نمود. اما تغییر مهمی‌که در این جا مدنظر است اضافه شدن عناصر جدید audio، video و canvas می‌باشد. عناصر audio و video همان طور که از نامشان معلوم است در جهت همان استانداردسازی فرمت‌های صوتی و تصویری گفته شده ایجاد شده اند و فایل‌های صوتی و تصویری را در خودشان حمل می‌کنند به گونه‌ای که مرورگران وب بتوانند آن‌هارا اجرا کنند، لازم به توضیح است که حالا در این استاندارد جدید وظیفه پشتیبانی از صوت و تصویر به عهده مرورگر قرار گرفته و نقش افزونه‌ها حذف شده است.

اما عنصر اصلی مورد توجه ما Canvas می‌باشد. عنصر canvas تکنولوژی بسیار جدیدی را به وب اضافه کرد که قابلیت ترسیم اشکال در یک سند وب را می‌دهد. قابلیت‌های این تکنولوژی را شاید بتوان با قابلیت‌هایی

مقایسه کرد که Flash در اختیار می‌گذاشت. یعنی ترسیم اشکال، ایجاد انیمیشن و همچنین محیط‌های تعاملی از جمله بازی‌ها. Canvas در واقعی عنصری در یک سند HTML است که دارای مشخصاتی از جمله طول و عرض می‌باشد. با این مشخصات یک محیط در اختیار قرار می‌گیرد که می‌توان با یک رابط کاربری از طریق JavaScript به آن دسترسی پیدا کرد و به ترسیم اشکال و ایجاد پویانمایی در ان پرداخت، همچنین به دلیل استفاده از script برای کار با آن امکان تعاملی کردن محیط نیز وجود داشته و به این صورت می‌توان به طراحی بازی نیز در آن پرداخت. شاید بتوان گفت که یکی از انگیزه‌های اساسی در ایجاد عنصر canvas به وجود آوردن جایگزینی برای Flash بوده تا با مقابله با مشکلاتی که افزونه‌های Flash داشتند بتوان امکانات آن‌ها را در محیط وب حفظ کرد به طوریکه بنای canvas از یکی از تکنولوژی‌های ایجاد شده در شرکت Apple بوده که در واقع با حذف Flash از مرورگر Safari خود این عنصر را ابتدا معرفی کرده بود.

با توجه به این که عنصر canvas در محیط به نسبت بسیار جدید است و امکانات بسیار نویی را در اختیار می‌گذارد بسیاری از کسانی که به کار پیاده‌سازی در محیط وب مشغول بوده اند شروع به کار با این عنصر و تست کردن آن کردند، در این میان انیمیشن‌ها و بازی‌های متنوع و بسیاری ایجاد شد تا معلوم شود قدرت این ابزار تا چه حد می‌باشد، از مهمترین پیاده‌سازی‌های موجود می‌توان به پورت کردن بازی Quake توسط شرکت گوگل بر روی canvas و البته با استفاده از کتابخانه‌های گرافیکی دیگر اشاره کرد. از آنجا که این بازی سه بعدی می‌باشد مشخص شد که قدرت این ابزار تا چه حد بالاست.

در میان بازی‌های ایجاد شده در canvas می‌توان یک روند تکاملی را دید که در طول این مدت وجود داشته است، بازی‌ها به تدریج غنی‌تر و دارای شکل‌های پیچیده تری شده اند، و در این روند یکی از کمبودهایی که به نظر می‌آید بسیار تعیین کننده می‌باشد نبود یک موتور فیزیک خوب در این محیط است. بازی‌های ویدئویی می‌توانند بدون موتور فیزیک هم پیاده شوند اما بازی‌هایی که نیاز داشته باشند تا پدیده‌های فیزیکی و

در طبیعت را در محیط کامپیوتر شبیه‌سازی کرده و تصور آن‌هارا نشان دهند نیاز دارند که یا خود یک موتور فیزیک در درون بازی خود ایجاد کنند و یا از یک موتور فیزیک دیگر استفاده کنند. با توجه به ثابت بودن پدیده های فیزیکی در دنیای واقعی و نیازهای مشابهی که این گونه بازی‌ها دارند می‌توان نتیجه گرفت که طراحی و ایجاد یک موتور فیزیک جدا می‌تواند به پیاده‌سازی طیف وسیعی از بازی‌های ویدئویی کمک کند، چنان که این بازی‌های می‌توانند بدون نیاز به نوشتن یک موتور فیزیک درونی برای خودشان جهت شبیه‌سازی پدیده‌هایی که می‌خواند در بازی خود به پدیده‌های فیزیکی شبیه باشد از این موتور فیزیک استفاده کنند. پس روشن می‌شود که چگونه در این محیط جدید ایجاد یکی موتور فیزیک که مختص این محیط است به کمک سیر پیشرفت پیاده‌سازی‌ها در این محیط می‌انجامد.

در نتیجه در این پروژه قصد داریم پدیده‌های فیزیکی دو بعدی قابل طرح در یک بازی را بررسی کرده و مدل‌های ریاضی آن‌ها را استخراج کنیم و با کمک آن‌ها پیاده‌سازی‌های مناسبی برای هر پدیده فیزیکی انجام دهیم و درنتیجه یک موتور فیزیک دو بعدی ایجاد کنیم. مهمترین مسئله این موتور فیزیک زبان پیاده‌سازی آن است که JavaScript می‌باشد، زبانی اسکریپتی که قرار است در محیط مرورگر اجرا شود و در برنامه‌هایی به کار آید که از canvas برای کشیدن شکل‌های وابسته به پدیده‌های فیزیکی استفاده می‌کنند. در واقع چون برای کار با canvasلازم است که با JavaScript کار شود، پیاده‌سازی موتور به این زبان ضروری می‌نماید.

فصل اول: تکنولوژی‌های بستر

۱.۱ مقدمه

برای پیاده‌سازی موتور فیزیک نیاز داریم تا محیطی که قرار است بازی‌های استفاده کننده در آن ایجاد شوند را به خوبی بشناسیم. در این فصل به مطالعه تکنولوژی‌های بستر این پروژه در وب پرداخته و امکانات، قابلیت‌ها و رابط‌های کاربری مربوطه را بررسی می کنیم تا بتوانیم با توجه به آن‌ها موتور فیزیک مناسب را بر پایه تکنولوژی‌های هدف بنا کنیم.

۲.۱ تکنولوژی‌ها

HTML یک زبان نشانه‌گذاری است که اسناد را با توجه به موقعیت و نوع عناصرشان استانداردسازی می‌کند، هر سندی که در محیط وب بخواهد قرار بگیرد و در مرورگر نشان داده شود لازم است که بازبان HTML نشانه‌گذاری شود. یک سند HTML نوعی به این صورت می‌باشد:

```
<! DOCTYPE html>
<html>
 <head>
  <title>Sample page</title>
 </head>
 <body>
  <h1>Sample page</h1>
  <p>This is a <a href="demo.html">simple</a> sample.</p>
  <!-- this is a comment -->
 </body>
</html>
```

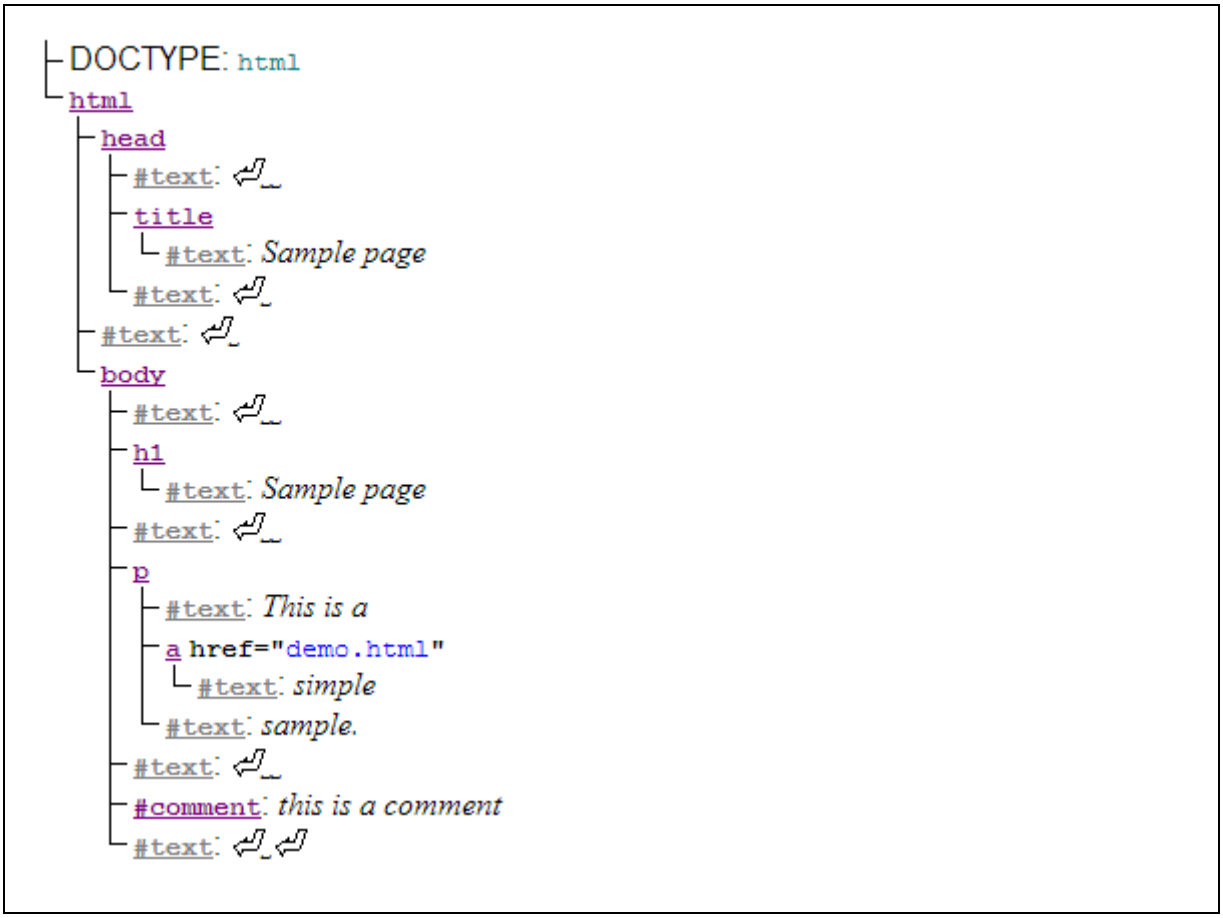
اسناد HTML شامل یک درخت از عناصر و متن می‌باشد. هر عنصر در منبع سند با یک برچسب شروع‌کننده شروع شده و با یک برچسب پایانی خاتمه می‌یابد:

```
<title>Sample page</title>
```

عناصر همچنین می‌توانند مشخصاتی برای خود داشته باشند که attribute نامیده می‌شود و از اسم و مقدار تشکیل شده است:

```
<input name=address disabled>
```

مرورگران وب به عنوان برنامه‌های کاربری این نشانه‌گذاری‌ها را مرور کرده و آن را تبدیل به درخت DOM (Document Object Model) می‌کنند. درخت DOM یک نمایش در حافظه از سند است. این درخت گره‌های گوناگونی دارد. مثالی از این درخت از کد بالا را در زیر مشاهده می‌کنید:



عنصر ریشه این درخت همیشه html می‌باشد و عناصر دیگر با توجه به ترتیب آمدنشان در درخت قرار می‌گیرند. درخت DOM می‌تواند توسط scriptها مورد دسترسی قرار بگیرد و دستکاری شود. اسکریپت‌ها (مثل JavaScript) برنامه‌هایی هستند که در درون برچسب <script> قرار می‌گیرند و یا توسط کنترل‌کننده‌های رخداد در مشخصات عناصر استفاده می‌شوند. برای مثال یک نوع از این اسکریپت‌نویسی را در زیر می‌بینید:

```
<form name="main">
```

```
 Result: <output name="result"></output>
 <script>
  document.forms.main.elements.result.value = 'Hello World';
 </script>
</form>
```

هر عنصر در درخت DOM توسط یک شی نشان داده می‌شود. و هر شی یک رابط کاربری برای خود دارد تا بتوان آن را دستکاری کرد. مثالی از این دستکاری:

```
var a = document.links[0]; // obtain the first link in the document
a.href = 'sample.html'; // change the destination URL of the link
a.protocol = 'https'; // change just the scheme part of the URL
a.setAttribute('href', 'http://example.com/'); // change the content attribute directly
```

HTML در واقع یک زبان فارغ از رسانه بیان آن است، رسانه آن هرچیزی می‌تواند باشد و محتوای آن می‌تواند در صفحه نشان داده شود و یا خوانده شود. برای مشخص کردن نحوه نشان داده محتوای سند HTML می‌توان از یک زبان استیل گذاری مانند CSS استفاده کرد. نمونه‌ای از زبان CSS را در زیر می‌بینید:

```
background: navy; color: yellow;
```

HTML5 به عنوان پنجمین نسخه از زبان نشانه‌گذاری گستره جهانی وب که ابتدا تنها برای توصیف اسناد علمی‌به کار می‌رفت و بعدا کاربردهای گسترده تری به خود گرفت در این جهت ایجاد شد تا یک حوزه فراموش شده از وب را در بر گرفته و استانداردسازی کند. این حوزه در واقع برنامه‌های تحت وب می‌باشد. در نتیجه HTML5 علاوه بر بهبود مسائل قبلی در HTML4 تاکید خود را بر وارد کردن این حوزه به HTML دارد.

یکی از مهمترین عناصر اضافه شده در HTML در نسخه پنجم عنصر Canvas می‌باشد که بنای طراحی و پیاده‌سازی موتور فیزیک در این پروژه نیز استفاده از آن در محیط ایجاد شده توسط این عنصر به زبان JavaScript است. Canvas که لغتش به معنی بوم نقاشی است عنصری است که یک محیط و اسکریپت‌هایی برای ایجاد شکل با وابستگی به رزولوشن محیط فراهم می‌کند که می‌تواند برای رندر کردن گراف‌ها و نمودارها،

گرافیک بازی‌ها، شکل‌های هنری و دیگر تصاویر بصری در لحظه استفاده شود. این اشکال می‌توانند توسط اسکریپت‌ها در Canvas به طور پویا ایجاد شوند. Canvas دو خصیصه عرض و طول دارد که در منبع سند HTML اندازه آن را در سند قابل ارائه توسط مرورگر نشان می‌دهد. برای دسترسی به رابط کاربری canvas لازم است که context مربوطه را از آن دریافت کنیم.

```
context = canvas. getContext(contextId [,... ])
```

آرگومان اول نشان دهنده نوع context مورد نظر ما و به تبع آن نوع رابط کاربری مورد نظرمان است. از جمله contextهای موجود می‌توان به رابط دو بعدی و یا webgl که یک رابط سه بعدی است اشاره کرد.

2D Context در واقع رابط مورد نظر ما برای ترسیم اشکال دو بعدی در canvas است. این زمینه اشیا, متدها و مشخصه‌هایی را برای ترسیم و دستکاری اشکال در سطح canvas فراهم می‌کند. این زمینه روش‌های گوناگونی را برای کار با اشکال ترسیمی فراهم می‌کند که به این قرارند:

یک پشته برای نگهداری وضعیت ترسیم‌ها که شامل ماتریس‌های تبدیل، محیط بریده شده و مقدار تمام پارامترهای گرافیکی موجود از جمله:

```
strokeStyle, fillStyle, globalAlpha, lineWidth, lineCap, lineJoin, miterLimit, shadowOffset
X, shadowOffsetY, shadowBlur,shadowColor, globalCompositeOperation, font, textAlign, t
extBaseline.
```

همچنین دو متد برای ذخیره و بازیابی این اطلاعات موجودند:

```
context. save()
context. restore()
```

که به مانند یک پشته عمل کرده و تمام اطلاعات را ذخیره و آخرین اطلاعات را بازیابی می‌کنند.

اشیا استایل در زمینه، نوع و خصوصیات ظاهری هر عنصری را مشخص می‌کنند که از آن‌ها می‌توان موارد زیر را نام برد:

```
LineStyles, Text styles, …
```

نمونه‌ای از انواع استایل‌هایی که بر روی نوشته می‌توان انجام داد را در شکل زیر مشاهده می‌کنید.

شکل ۱ تنظیمات نوشته در canvas

مسیرها پایه ای‌ترین عنصر ترسیم اشکال در زمینه هستند. هر مسیر از یک یا چند زیر مسیر تشکلی شده است و هر زیر مسیر از یک یا چند نقطه که توسط خط‌های مستقیم و یا خمیده به هم متصل شده اند و همچنین یک وضعیت برای این که مشخص شود آیا مسیر بسته است یا خیر. مسیر بسته مسیری است که آخرین نقطه آن به اولین نقطه آن با یک خط مستقیم متصل می‌شود. برای مثال برای ترسیم یک مسیر می‌توان از کد زیر استفاده کرد:

```
context. moveTo(x, y)
path. moveTo(x, y)
```

و برای بستن آن:

```
context. closePath()
```

```
path. closePath()
```

برای ترسیم یک خط مستقیم تا نقطه دلخواه:

```
context. lineTo(x, y)
path. lineTo(x, y)
```

برای ترسیم یک خط خمیده Bezier تا نقطه دلخواه با نقطه کنترلی دلخواه:

```
context. quadraticCurveTo(cpx, cpy, x, y)
path. quadraticCurveTo(cpx, cpy, x, y)
```

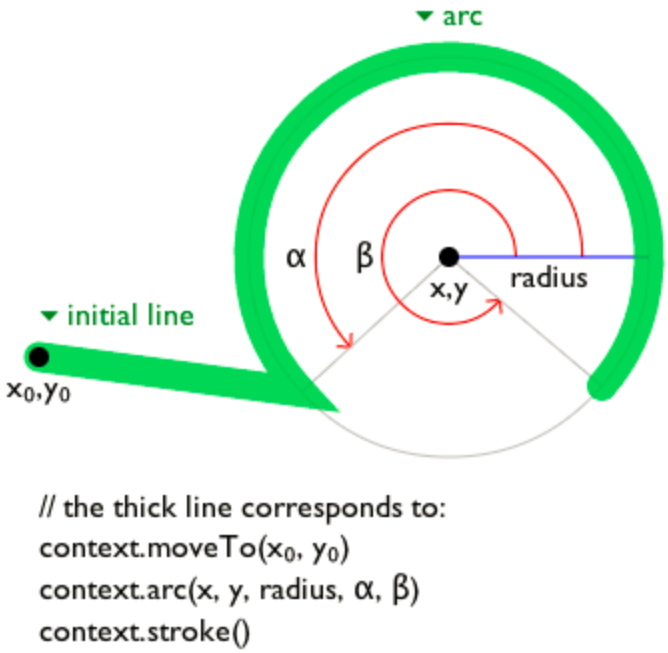
همچنین یک خط خمیده Bezier با دو نقطه کنترلی:

```
context. bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
path. bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

و یک خط خمیده دایره‌ای با شعاع معلوم:

```
context. arcTo(x1, y1, x2, y2, radius)
path. arcTo(x1, y1, x2, y2, radius)
```

یک مثال از یک مسیر دوقسمتی که یک خط مستقیم و یک خط خمیده را دارد در شکل زیر مشاهده می‌کنید:



شکل ۲ کشیدن خط در canvas

برای ترسیم یک مستطیل نیز می‌توان از دستورات زیر بهره برد:

```
context. rect(x, y, w, h)
path. rect(x, y, w, h)
```

اشیا نوع Path برای اعلان مسیرها به کار می‌روند. و توابعی برای دستکاری آنها از جمله ترکیبشان یا نوشتن کلمات دارند:

```
addPath(b, transform)
addPathByStrokingPath(b, styles, transform)
addText()
addPathByStrokingText()
addText()
addPathByStrokingText()
```

هر شی در زمینه دارای یک ماتریس تبدیل است که در ابتدای ایجاد آن شی این ماتریس یکه می‌باشد.

برای دستکاری این ماتریس و تبدیل اشیا متدهای زیر را در اختیار داریم:

```
context. scale(x, y)
context. rotate(angle)

context. translate(x, y)

context. transform(a, b, c, d, e, f)

context. setTransform(a, b, c, d, e, f)
```

همچنین توابع مختلفی نیز برای تغیر استایل اشیا وجود دارد:

```
context. fillStyle [ = value ]

context. strokeStyle [ = value ]

pattern = context. createPattern(image, repetition)
```

یکی دیگر از اشکال کشیدنی پایه‌ای در canvas مستطیل‌ها هستند، با توابع زیر می‌توان آن‌ها را رسم کرد:

```
context. clearRect(x, y, w, h)

context. fillRect(x, y, w, h)

context. strokeRect(x, y, w, h)
```

همچنین برای ترسیم نوشته‌ها در canvas:

```
context. fillText(text, x, y [, maxWidth ] )

context. strokeText(text, x, y [, maxWidth ] )
```

در رابطه با مسیرها بعد از مشخص شدن شکل آن‌ها می‌توان با توابع زیر آن‌ها را ترسیم کرد:

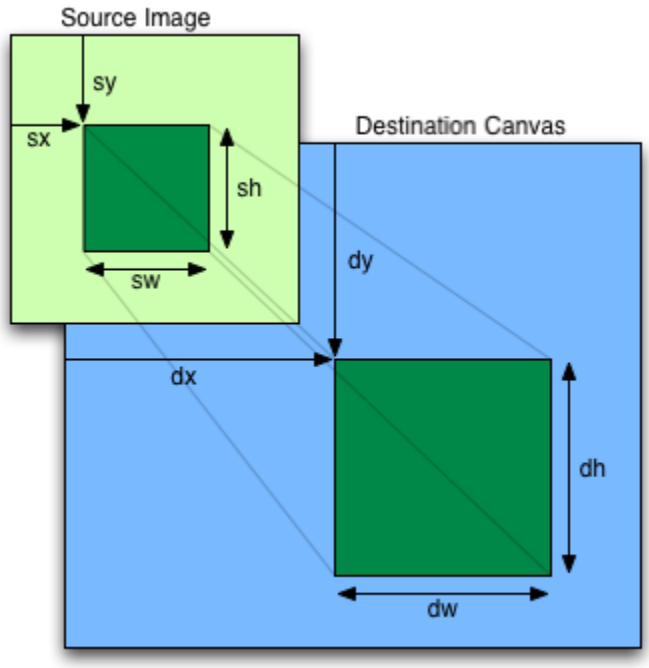
```
context. beginPath()
context. fill()
context. fill(path)
context. stroke()
context. stroke(path)
```

```
context. drawSystemFocusRing(element)
context. drawSystemFocusRing(path, element)
context. scrollPathIntoView()
context. scrollPathIntoView(path)
context. clip()
context. clip(path)
context. isPointInPath(x, y)
context. isPointInPath(path, x, y)
```

یکی دیگر از قابلیت‌های اساسی canvas توانایی ترسیم عکس‌ها در آن می‌باشد، برای آین کار از توابع زیر می‌توان استفاده کرد:

```
drawImage(image, dx, dy)

drawImage(image, dx, dy, dw, dh)

drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
```

آرگومان‌های فوق با توجه به شکل زیر تفسیر می‌شوند:



شکل ۳ تصویر در canvas

در زمینه به طور کلی یکسری از نواحی به نام نواحی Hit Region یا نواحی برخورد در یک لیستی به نام hitRegions قرار می‌گیرند. هر Hit Region شامل این موارد است: یک مجموعه‌ای از پیکسل‌های bitmap در canvas که این ناحیه برخورد در آن قرار می‌گیرد، یک محیط مرزی در bitmap در canvas که محیط ناحیه برخورد را در بر می‌گیرد، و همچنین چند خصوصیت انتخابی از جمله شکل curser در آن ناحیه و غیره. با دستورات زیر این نواحی می‌توانند به وجود آمده و از بین بروند:

```
context. addHitRegion(options)

context. removeHitRegion(options)
```

که در آن‌ها Options می‌تواند از قسمت‌های زیر تشکیل شود:

```
path (default null)
id (default empty string)
parentID (default null)
cursor (default "inherit")
control (default null)
label (default null)
role (default null)
```

این نواحی برخورد کاربردهای مختلفی می‌توانند داشته باشند که از آن جمله می‌توان به راحتتر کردن تشخیص برخوردها با ماوس اشاره کرد. همچنین این نواحی می‌توانند در بازی‌ها برای بست کردن به موتور فیزیک جهت تشخیص برخورد به کار بیایند.

دست آخر نیز زمینه دوبعدی ابزارهایی را هم برای دستکاری پیکسلی در اختیار می‌گذارد تا چنان که ابزارهای دیگر کافی نبودند بتوان از آن‌ها بهره برد تا هر شکل دلخواهی را ترسیم نمود. برای این کار باید از متدهای زیر استفاده کرد:

```
imagedata = context. createImageData(sw, sh)

imagedata = context. createImageData(imagedata)
```

```
imagedata = context. getImageData(sx, sy, sw, sh)
imagedata. width
imagedata. height
imagedata. data
context. putImageData(imagedata, dx, dy [, dirtyX, dirtyY, dirtyWidth, dirtyHeight ])
```

همچنین تمام اشکال در زمینه می‌توانند دارای سایه و ضریب آلفای جداگانه‌ای باشند:

```
context. globalAlpha [ = value ]
context. globalCompositeOperation [ = value ]
context. shadowColor [ = value ]
context. shadowOffsetX [ = value ]
context. shadowOffsetY [ = value ]
context. shadowBlur [ = value ]
```

همان طور که قبلا هم اشاره شد، 2D Context زمینه اصلی مورد نظر ما در این پروژه جهت پیاده‌سازی موتور فیزیک به هدف آن است، و به همین جهت توضیحات کلی در رابطه با ویژگی‌ها و ابزارهای موجود برای کار با آن و همچنین رابط کاربری اش داده شد. اما زمینه دیگری که در این محیط برای canvas وجود دارد و قابل ذکر است زمینه webgl می‌باشد، این زمینه یک رابط کاربری بسیار نزدیک و در رابطه با OpenGL که خود یک رابط کاربری بین زبانی و چند سکویه برای ارائه گرافیک دو بعدی و سه بعدی می‌باشد ارائه می‌کند. اما webgl به طور کلی رابط کاربری طراحی سه بعدی گرافیک در محیط وب بعنی canvas می‌باشد. توضیح این رابط کاربری خارج از حوزه این گزارش بوده و تنها به همین اشاره اکتفا می‌شود.

۳.۱ جمع‌بندی و نتیجه‌گیری

در این فصل به بررسی کلی امکانات HTML5 و canvas و همچنین 2D Context پرداختیم و نشان دادیم که چطور این امکانات می توانند در جهت ایجاد گرافیک‌های برداری در محیط یک سند وب به کار گرفته شوند. این امکانات همچنین نشان می دهند که چطور می توان یک بازی را در این محیط ایجاد کرد. آنچه در این فصل گفته شد مروری کلی بر این موضوع بود و بیشتر جهت نشان دادن بستر ایجاد موتور فیزیک مطرح شد.

در فصل آینده به بررسی ماهیت فیزیک در بازی و مطالعه انواع و شکل‌های آن و همچنین توضیح این مطلب خواهیم پرداخت که در این پروژه فیزیک به چه تحوی پیاده‌سازی خواهد شد.

فصل دوم: بررسی فیزیک بازی

۱.۲ مقدمه

در بازی‌های ویدئویی اغلب لازم است تا شرایط و یا قسمتی از شرایط دنیای واقعی شبیه‌سازی شود، این شبیه‌سازی به این معنی است که بازیکن حس کند آنچه در بازی می‌بیند می‌تواند ما به ازا جسم و شی واقعی در دنیای بیرون باشد، به این صورت حس واقعی‌تر کردن بازی به هیجان و جذابیت بازی به شدت کمک خواهد کرد.

فیزیک نظام بسیار بزرگی است و فیزیک آکادمیک صدها زیر رشته دارد که هر کدام جنبه‌هایی از دنیای فیزیکی را بیان می‌کنند برای مثال نحوه کارکرد نور و یا یک انفجار اتمی.

اما بعضی قسمت‌های فیزیک در بازی کاربرد می‌تواند داشته باشد. مثلا مبحث نور شناسی برای درک حرکت نور، بازتاب‌های آن بر روی جسم و در نتیجه ایجاد گرافیک‌های زیبا بسیار کاربرد دارد که به آن شیوه Way-Tracing گفته می‌شود اما این مبحثی از فیزیک نیست که در بازی‌های کامپیوتری بتواند کاربرد داشته باشد و چون پردازشی بسیار کندی دارد در این بحث جای نمی‌گیرد. یا برای مثال کاربردی از شبیه‌سازی یک نیروگاه اتمی‌در بازی و فیزیک آن نمی‌توان متصور بود و در نتیجه در فیزیک بازی جای نمی‌گیرند.

با این بحث معلوم می‌شود که هر آنچه در فیزیک هست در بازی کاربرد ندارد و در واقع وقتی صحبت از فیزیک بازی می‌کنیم منظور بیشتر از همه مکانیک کلاسیک است: قوانینی که حرکات اجرام بزرگ در نتیجه نیروهایی چون جاذبه را تعیین می‌کنند. در فیزیک آکادمیک این موضوعات با موضوعاتی چون کوانتوم و نسبیت جایگزین شده اند اما در دنیای بازی این قوانین می‌توانند به کار گرفته شوند تا بتوان حس صلب بودن اجسام، با جرم، اینرسی، ارتجاع و شناور بودن را القا کرد.

فیزیک بازی در ابتدایی‌ترین بازی‌ها تا به امروز موجود بوده است و در حرکات ذرات در بازی‌ها تجلی داشته است: جرقه‌ها، آتش‌بازی، بالستیک گلوله‌ها، دود و انفجارها. در مراحل پیشرفته‌تر فیزیک بازی برای سه

دهه است که در شبیه‌سازی پرواز به کار رفته است. همچنین به فیزیک ماشین‌ها با توجه به تایر، شکل، تعلیق و موتورهای مختلف اشاره کرد.

همان طور که قدرت پردازشی بالا رفت، جابجایی جعبه‌های زیادی که روی هم قرار می‌گیرند، دیوارهایی که تخریب می‌شوند و به تکه‌های خود می‌شکنند نیز قابل شبیه‌سازی شدند که در حوزه فیزیک جسم صلب قرار می‌گیرند. لباس‌ها، پرچم‌ها و طناب‌ها بعدا از جمله مسائل فیزیک جسم نرم می‌باشند که شبیه‌سازی شدند.

در این پروژه هدف پیاده‌سازی یک موتور فیزیک ذره‌ای می‌باشد که با بررسی قوانین فیزیکی آن به نحوه پیاده‌سازی آن می‌پردازیم.

۲.۲ موتور فیزیک

در بازی‌های کامپیوتری پدیده‌های فیزیکی آن چنان که نیاز هستند می‌توانند در خود بازی پیاده شوند و بدون نیاز به یک موتور جداگانه از پس قوانین لازم بر بیایند. اما این موضوع تنها به شرطی صحیح است که تعداد پدیده‌های فیزیکی مورد نظر کم باشد، اگر تعداد پدیده‌های فیزیکی که در بازی می‌خواهیم به آن‌ها بپردازیم زیاد باشد، پیاده‌سازی آن‌ها در داخل بازی غیر عملی است و لازم است که حتما آن‌ها را خارج از بازی طوری پیاده کرد که پیاده‌سازی بازی بتواند از توابع محاسبه‌گر پدیده‌های فیزیکی‌شان استفاده کند، در این حالت یک موتور فیزیک به وجود آمده که اگر به درستی مستقل از بازی پیاده شود می‌تواند برای هر بازی دیگری که نیاز به پیاده‌سازی پدیده‌های فیزیکی دارد به کار گرفته شود و این موضوع سبب شده است که موتورهای فیزیک عمومی به وجود بیاید که بازی‌های مختلفی بتوانند از آن استفاده کنند. وجود یکی موتور فیزیک جداگانه که بتوان در یک بازی دلخواه از آن استفاده کرد سازنده بازی را از بسیاری از مسائل و پیچیدگی

ها رها کرده و فرصت بیشتری را در اختیار او قرار می‌دهد، چنان که موتورهای فیزیک در طول زمان چنان قدرتمند می‌شوند که پیاده‌سازی جداگانه آن‌ها برای هر بازی اصلا مقرون نخواهد بود.

پس می‌توان این طور نتیجه گرفت که موتور فیزیک یک قطعه کد است که همه چیز در مورد فیزیک را می‌داند و با هیچ وابستگی به سناریوی خاصی از بازی پیاده شده است. اساسا پدیده‌های مشابه فیزیکی بسیاری هستند که می‌توانند برای مثال از یک ویژگی عمومی‌در یک موتور فیزیک استفاده کنند تا شبیه‌سازی شوند اما برای هر پدیده جداگانه لازم است تا مشخصات آن به موتور داده شود تا برای آن پدیده خاص شبیه‌سازی را انجام دهد.

به یک دید دیگر در این نتیجه می‌توان گفت که موتور فیزیک در واقع یک ماشین حساب بزرگ و بسیار قدرتمند است که تمام محاسبات ریاضی پدیده‌های فیزیکی را انجام می‌دهد اما اطلاعی از این ندارد که این محاسبات دقیقا برای چه چیزی است.

لازم به ذکر است که محاسبات اشاره شده باید در نتیجه دریافت داده‌هایی از طرف خود بازی باشد، که این اطلاعات منجر به محاسبات و شبیه‌سازی مورد نظر می‌شود، دریافت و دادن این اطلاعات از و به موتور خارج از مبحث موتور است و به پیاده‌سازی بازی مربوط می‌شود.

دو فایده بزرگ در نتیجه استفاده از موتور بازی وجود دارد.

اول صرفه جویی بسیار زیاد در زمان است. یک موتور فیزیک شامل هزاران خط کد می‌باشد، پیاده‌سازی هر باره آن‌ها برای هر بازی منطقی نمی‌نماید و استفاده از یک موتور فیزیک جداگانه کاملا مقرون به صرفه است.

دوم کیفیت است. هر پدیده فیزیکی در یک موتور اختصاصی به بهترین نحو ممکن می‌تواند پیاده شود و از آن‌ها استفاده شود. پیاده‌سازی مجدد و مجدد ان‌ها حتی اگر خوب هم بشود پیاده کرد معلوم نیست که هر باز

این کیفیت بتواند حفظ شود. در ضمن شاید بتوان پدیده‌های مختلف را در جای خود در یک بازی پیاده‌سازی کرد اما وقتی این پدیده‌ها بخواهند با هم ترکیب شوند و انواع حرکات در محیط‌های ترکیبی را انجام دهند آنگاه بدون وجود یک موتور فیزیک که هر کدام از پدیده‌ها را جداگانه بررسی می‌کند این کار عملی نخواهد بود.

۳.۲ نوع و روش پیاده‌سازی موتور فیزیک

برای پیاده‌سازی در این پروژه از میان انواع روش‌هایی که موتور فیزیک را می‌توان طراحی و پیاده‌سازی کرد، گزینه‌های موجود را بررسی کرده و بهترین گزینه با توجه به هدف و امکانات موجود انتخاب شده است.

در اولین مسئله باید مشخص شود که موتور آیا یک موتور اجسام صلب کامل باید باشد و یا یک موتور تجمیع جرم. موتور اجسام صلب اجسام را به طور کامل و کلی در نظر گرفته و حرکات و چرخش آن‌ها را پیاده می‌کنند. به مانند یک جعبه که به طور کامل جسمش می‌تواند شبیه‌سازی شود. موتور تجمیع جرم اجسام را به عنوان تعداد زیادی از اجرام ذره در نظر گرفته و با آن‌ها به این صورت بر خورد می‌کند، در واقع محاسبات برای هر ذره و در نتیجه کل جسم انجام می‌شود. یک جعبه می‌تواند ۴ ذره باشد که در گوشه آن به وسیه میله‌هایی به هم متصل شده‌اند.

موتور تجمیع اجرام از نظر پیاده‌سازی دارای این مزیت است که نیازی به دانش چرخش اجسام ندارد، اما می‌توان یک موتور تجمیع اجرام را با گسترش دادن به یک موتور اجسام صلب تبدیل کرد و بدین صورت از مزایای آن هم بهره‌مند شد. در این پروژه موتور پیاده‌سازی شده، تجمیع اجرام است که می‌تواند بعدا به اجسام صلب تبدیل شود.

در گام دوم باید نحوه تحلیل برخوردها مشخص شود. این که چگونه و با چه روشی محل برخوردها شناسایی، تحلیل و اقدام آن‌ها انجام شود.

یک روش بررسی کردن و تحلیل هر برخورد به طور جداگانه و یکی یکی است. این روش، روش تکرار نامیده می‌شود که مهمترین مزیت آن سرعت آن است. یک برخورد به سرعت تحلیل می‌شود و تعداد زیادی برخورد در زمان کمی‌به این صورت می‌توانند تحلیل شوند.

یک روش دیگر که Jacoban-based نام دارد، تمام برخوردها را با هم در نظر گرفته و یک تحلیل کلی روی آن‌ها انجام می‌دهد. این روش بسیار زمان گیر است و بیشتر در شبیه‌سازی‌های فیزیکی آکادمیک به کار می‌رود.

به همین دلیل در این موتور از روش اول که سریع‌تر است و مارا به نتیجه دلخواهمان می‌رساند استفاده می‌کنیم.

در گام بعد باید نوع ایجاد محرک حرکتی در موتور بررسی کنیم، در واقع در پیش روی ما سه راه است که تمام نیروها را همان طور که در طبیعت وجود دارند شبیه‌سازی کنیم، و یا این که آن نیروهایی که در زمان بسیار کوتاهی وارد می‌شوند را به صورت تکانه[1] و نه نیرو شبیه‌سازی کنیم و یا این که تمام نیروها را در واقع به صورت تکانه‌ها پیاده‌سازی کنیم. در این پروژه روش دوم انتخاب شده است چرا که به حس واقعی از طبیعت بسیار نزدیک‌تر است، به این صورت نیروهایی که به مدت زیادی وارد می‌شوند را به صورت همان نیرو در نظر گرفته اما نیروهایی که به مدت کمی وارد می‌شوند را به صورت تکان و ضربه در نظر گرفته و این گونه آن‌ها را محاسبه می‌کنیم.

۴.۲ جمع‌بندی و نتیجه‌گیری

در این فصل بررسی کردیم که موتور فیزیک در واقع چیست و در دنیای بازی‌سازی به چه استفاده‌ای می آید و گفتیم که این موتور در واقع به طور کلی مسئولیت شبیه‌سازی پدیده‌های فیزیک در دنیای واقع را

[1] impulse

دارد و از انواع مختلف آن در این پروژه به پیاده‌سازی موتور تجمیع اجرام که یک موتور شبیه‌سازی با کمک ذرات بدون جهت است می پردازیم.

در فصل بعد اولین قدم در را ه ساخت این موتور یعنی ایجاد فیزیک ذرات را برخواهیم داشت و با مطالعه جنبه‌های مختلف یک ذره، آن را پیاده‌سازی خواهیم کرد.

فصل سوم: فیزیک ذرات

۱.۳ مقدمه

موتور فیزیک ذرات در واقع موتوری است که اساس کار آن ذرات فیزیکی هستند. این ذرات دارای موقعیت، سرعت، شتاب، نیرو و غیره می باشند. بررسی این ذرات و به روز رسانی آن‌ها وظیفه اصلی موتور بوده و هسته مرکزی موتور را تشکیل می دهد. در این فصل به بررسی ریاضیات آن‌ها پرداخته و پیاده‌سازی شان را مطرح می کنیم.

۲.۳ ریاضیات ذرات

بردار یک عنصر از فضای برداری است، ساختاری که خصوصیات ریاضی خاصی برای اعمال ریاضی دارد. در این پروژه بردارهای دوبعدی در فضای معمول دوبعدی (Euclidean) مورد بحث هستند. در این حالت هر بردار نشان دهنده یک موقعیت در این فضا می‌باشد. مهمترین کاربرد بردارها در مشخص کردن مختصات در فضای دو بعدی و سه بعدی می‌باشد. این مختصات می‌تواند توسط دو مقدار مختصاتی که هر کدام فاصله‌ای ثابت از یک محور مختصاتی هستند نشان داده می‌شود. این یک سیستم مختصاتی کارتزین می‌باشد.

هر بردار یک نقطه در صفحه را نشان می‌دهد و هر نقطه در فضا به یک و تنها یک بردار مرتبط است. در موتور فیزیک این پروژه از آن جا که یک بردار در واقع می‌تواند مشخصات بک نقطه در صفحه نیز باشد، کلاس مربوط به آن Point نامیده می‌شود.

نکته مهم دیگر این است که بردارها همچنین می‌توانند نشان دهنده جابجایی و فاصله بین نقاط صفحه باشند و درنتیجه عملیات‌های جمع، ضرب و تفریق برای بردارها این گونه دارای معنی می‌شوند.

در این پروژه عملات‌های گوناگونی بر رور بردارها انجام می‌شود که از آن جمله می‌توان به نرمال کردن بردارها، دریافت طول آنها، زاویه آن‌ها، عملیات ریاضی جمع، تقریق و ضرب‌های داخلی و خارجی اشاره کرد.

در این پروژه ما تغییرات مشخصات ذرات در طول زمان را حساب می‌کنیم و از آن جمله و پایه‌ترین تغییر، تغییر موقعیت ذره در طول زمان بر اساس سرعت، شتاب، نیروهای وارده و غیره می‌باشد. از دو دید این تغییرات را می‌توان بررسی کرد. اول نحوه تغییر این متغیرها می‌باشد و دوم نتیجه تغییر آن‌ها. در مورد نحوه تغییر آن‌ها باید به حساب دیفرانسیل رجوع کرد و نتیجه آن‌ها از طریق حساب انتگرال و جمع کننده‌ها میسر خواهد بود.

در حساب دیفرانسیل به این موضوع می‌پردازیم که تغییرات متغیرها در طول زمان چگونه است، این تغییر سرعت است. لازم به ذکر است که سرعت دارای دو معنی velocity و speed می‌باشد، که در واقع speed تنها به اندازه سرعت اشاره دارد و velocity برداری است که جهت سرعت و به عبارت دیگر تغییر را نیز مشخص می‌کند. بدین ترتیب از معادله زیر می‌توانیم مفهوم سرعت را به روشنی ببینیم:

$$v = \frac{p' - p}{\Delta t} = \frac{\Delta p}{\Delta t}$$

که در آن p موقیعیت قبلی، p' موقعیت جدید و v سرعت است و از آن جایی که ما به سرعت در کوتاه ترین زمان ممکن علاقه داریم و نه سرعت متوسط در یک زمان طولانی به معادله زیر می‌رسیم:

$$v = \lim_{\Delta t \to 0} \frac{\Delta p}{\Delta t} = \frac{\mathrm{d}p}{\mathrm{d}t}$$

این موضوع در مورد سرعت ذره کاملا روشن است اما در مورد شتاب هم می‌توان گفت که شتاب سرعت تغییر سرعت است و در نتیجه خواهیم داشت:

$$a = \lim_{\Delta t \to 0} \frac{\Delta v}{\Delta t} = \frac{\mathrm{d}v}{\mathrm{d}t}$$

و اگر به نسبت مکان بخواهیم شتاب را در نظر بگیریم:

$$a = \frac{\mathrm{d}v}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}\frac{\mathrm{d}p}{\mathrm{d}t} = \frac{\mathrm{d}^2p}{\mathrm{d}t^2}$$

لازم به ذکر است که تمام این معادلات برای بردارها نیز صادق می‌باشند چنان که هر کدام از دو مقدار آنها در این معادله‌ها قرار می‌گیرند و می‌توان a و p را بردارهای شتاب و سرعت نامید.

حساب انتگرال در نقطه مقابل حساب دیفرانسیل قرار می‌گیرد. پس ما با دانستن سرعت و موقعیت ذره در زمان خال می‌توانیم موقعیت آن را در آینده پیش بینی کنیم، این یک نکته اساسی در پیاده‌سازی فیزیک ذرات می‌باشد. این خاصیت برای به روز کردن موقعیت و سرعت ذره در هر لحظه به کار می‌رود. قطعه کدی که این کار را انجام می‌دهد Integrator نام دارد.

این خاصیت انتگرال در واقع بدون وجود انتگرال‌های پیچیده در ریاضیات و تنها با عملیات جبری قابل پیاده‌سازی است. نحوه محاسبه سرعت و موقعیت جدید به این صورت خواهد بود:

$$p' = p + \dot{p}t$$

$$\dot{p}' = \dot{p} + \ddot{p}t$$

البته موقعیت ذره در صورتی که دارای شتاب باشد از معادله زیر استخراج می‌شود:

$$p' = p + \dot{p}t + \ddot{p}\frac{t^2}{2}$$

و باز هم یادآوری می‌شود که این معادله‌ها همگی برای برای بردارها نیز صادق می‌باشند.

۳.۳ جمع‌بندی و نتیجه‌گیری

در این فصل ذرات را به عنوان قسمت اصلی موتور بررسی کردیم و ریاضیات آن‌ها را مطرح نمودیم. این ریاضیات بعدا در پیاده‌سازی فصل بعد که به همراه قانون‌های نیوتن مطرح می شوند به کار گرفته خواهند شد.

فصل چهارم: قوانین حرکت

۱.۴ مقدمه

موتورهای فیزیک بر پایه قوانین حرکت نیوتون بنا می‌شوند. در این پروژه از این قوانین استفاده می کنیم. سه قانون نیوتون توضیح می‌دهند که موقعیت یک نقطه جرم‌دار چگونه است. این نقطه جرم‌دار را ما در این موتور فیزیک particle می‌نامیم. در ادامه به بررسی پیاده‌سازی آن‌ها می پردازیم.

۲.۴ ذره

یک ذره، موقعیت دارد ولی جهت ندارد. به مانند یک گلوله و یه ذره نور هیچ نیاز به جهت یک ذره وجود ندارد ضمن این که در فیزیک ذراتی که بررسی می‌کنیم جهت ذره خارج از موضوع است. برای هر ذره باید مشخصه‌های گوناگونی را نگهداری کنیم: موقعیت کنونی، سرعت، شتاب و غیره. به این صورت شیئ particle را برای ذره با prototype زیر مشخص می‌کنیم.

```
function Particle(){
   this.position = new Point();
   this.velocity = new Point();
   this.acceleration = new Point();
}
```

۳.۴ دو قانون اول

در ابتدا تنها دو قانون اول را در نظر می‌گیریم که به این قرار می‌باشند:

یک شیئ با سرعت ثابت به حرکت خود ادامه می‌دهد مگر این که نیرویی به آن وارد شود.

نیرویی که به یک شیئ وارد می‌شود شتابی در آن ایجاد می‌کند که با جرم آن نسبت مستقیم دارد.

قانون اول بیان می‌کند که در صورت نبود هیچ نیرویی جسم چگونه عمل می‌کند و به حرکت خودش با همان سرعتی که دارد ادامه می‌دهد. سرعت ذره هیچگاه تغییر نمی‌کند و موقعیت ان بنا به آن سرعت دائما به روز می‌شود. اما در واقعیت روزمره این اتفاق نمی‌افتد و ذرات پس از مقداری حرکت می‌ایستند. این در واقع اثر

نیروی اصطکاک و یا مقاوت هوا (drag) می‌باشد. پس در واقع این نیروها به جز زمانی که در فضا حرکت کنیم همواره وجود دارند. ما نیروی drag را در فصل تولید کننده‌های نیرو بررسی خواهیم کرد اما به جز این نیرو می توانیم از یک نوع خاص نیروی مقاومت محیط در خود شئی particle بهره برد که در مواقع لازم می‌توان از آن استفاده کرد بدون این که یک نیروی اضافه لازم باشد در فیزیک تعریف کنیم. این مقاوت را damping می‌نامیم.

```
this.damping = null;
```

به این صورت با توجه به مقدار damping یک نسبتی از سرعت شیئ را در هر به روز رسانی در integrator کم می‌کنیم. اگر damping مقدار صفر را داشته باشد، بعد از update هیچ مقداری از سرعت باقی نخواهد ماند و مقدار یک به معنی این است که ذره هیچ مقدار از سرعتش را از دست نخواهد داد. به این صورت مقادیر نزدیک به یک برای شبیه‌سازی مقاومت محیط نزدیک به واقعیت هستند.

قانون دوم مکانیزم تاثیر نیرو را به ما می‌دهد، به این صورت که نیرو چیزی است که شتاب ذره را تغییر می‌دهد، بنابراین ما نمی‌توانیم کاری کنیم که موقعیت یا سرعت جسم مستیما تغییر کند، و باید با اعمال نیرو شتاب آن را تغییر دهیم تا به سرعت و یا موقعیت مورد نظر برسد. بنابراین برای این که پرش و یا اتفاقات غیر عادی فیزیک ذره نیفتد برای ایجاد تغییر در ذره موقعیت و سرعت آن را مستقیما تغییر نمی‌دهیم بلکه شتاب آن را تغییر می‌دهیم تا حرکتش حالت طبیعی داشته باشد.

بنابراین معادله دوم تعیین می‌کند که چگونه نیرو به شتاب ربط پیدا می‌کند. معادله نیرو و ارتباط آن به شتاب به این صورت می‌باشد:

$$f = ma = m\ddot{p}$$

و درنتیجه برای بدیت آوردن شتاب ناشی از نیروی وارده داریم:

$$\ddot{p} = \frac{1}{m} f$$

به طوری که f نیرو و m جرم ذره می‌باشد. و البته نیرو هم به مانند شتاب در این جا بردار است.

با توجه به این توضیحات ضروری می‌آید که جرم ذره را نیز به عنوان یک متغیر به سیئ particle|اضافه کنیم. اما این کار را به صورت مستقیم انجام نمی‌دهیم. به دلیل این که یک ذره هیچگاه نمی‌تواند جرم صفر داشته باشد و وجود داشتن جرم صفر با توجه به معادله اخیر منجر به خطای تقسیم بر صفر می‌شود و در واقع به معنی شتاب بی نهایت است و همچنین به این دلیل که جرم بی نهایت در موتور فیزیک می‌تواند دارای معنی باشد به این صورت که ذره جرم بسیار بزرگی دارد که حرکت نمی‌کند و هیچ محاسبات حرکتی برای آن لازم نیست، به مانند گیره‌هایی برای پاندول و یا دیوارها و مرزهای محیطی و در زبان JavaScript نمی‌توانیم یک عدد بی نهایت نشان دهیم پس به جای خود جرم جسم، معکوس جرم جسم را در particle قرار می‌دهیم، به این صورت هم امکان به وجود داشتن ذره با جرم صفر را از بین برده‌ایم و هم این که با صفر قرار دادن معکوس جرم توانسته‌ایم جرم بی نهایت را پیاده‌سازی کنیم:

```
this.inverseMass = null;
```

جاذبه یکی از مهمترین نیروهای موجود در طبیعت می‌باشد و بین هر دو جسم وجود دارد که به جرم آن‌ها و فاصله شان مربوط است. قانون جهانی گرانش که توسط نیوتون بیان شده است دارای معادله زیر است:

$$f = G\frac{m_1 m_2}{r^2}$$

که m جرم جسم‌ها، r فاصله آن‌ها و G ثابت جهانی گرانش است. این جاذبه بین اجسام کوچک قابل صرف نظر است و بین ما و زمین بسیار زیاد می‌باشد بنابراین چون ما تنها برای جاذبه زمین می‌خواهیم در موتور

جایی قرار دهیم از این معادله صرف نظر کرده و به معادله ساده‌تر زیر برای هر جسم در زمین می‌رسیم:

$$f = mg$$

که در آن g شتاب گرانش زمین است و از معادله قبلی به دست می‌آید:

$$g = G\frac{m_{\text{earth}}}{r^2}$$

بنابراین فارق از جرم جسم شتاب گرانش همواره ثابت خواهد بود. ما دو راه بنابراین برای اعمال جاذبه به ذرات در این موتور فیزیک داریم یکی اعمال مستقیم شتاب جاذبه به آن‌ها و دیگری ایجاد یک تولید کننده نیرو در موتور که در فصل خود مورد توضیح قرار می‌گیرد اما علیرغم این که در آخر در این موتور راه دوم در پیش گرفته خواهد شد اما فعلا تا زمان ایجاد تولید کننده‌های نیرو در مثال‌ها از روش اول استفاده می‌کنیم. مقدار g حتی می‌تواند برای هر ذره متفاوت باشد و به صورت برداری که مقدار y آن g– نشان داده شود.

۴.۴ اعمال معادلات

بنابر تمام توضیحاتی که تا کنون داده شد می‌توان حالا Integrator را برای particle پیاده‌سازی کرد. در هر فریم تصویر integrator شتاب هر ذره را بررسی می‌کند و عمل به روز رسانی‌ها را با توجه به آن انجام می دهد. در این جا integrator دارای دو قسمت است یکی به روز کردن موقعیت ذره با توجه به سرعتش و دیگری به روز کردن سرعت آن با توجه به شتابش. Integrator همچنین نیازمند یک پارامتر زمان است تا این به روز رسانی‌ها را بتواند انجام دهد. برای این کار مدت زمان سپری شده بین هر فریم را به آن داده تا محاسبات لازیم بین هر فریم را انجام دهد.

معادله به روز رسانی موقیعت ذره چنان که قبلا مشاهده کردیم به صورت زیر است:

$$p' = p + \dot{p}t + \frac{1}{2}\ddot{p}t^2$$

اما قسمت سوم معادله چنان که زمان بین هر فریم بسیار کم می‌باشد و چنانچه در خودش ضرب شود بسیار عدد کوچکی خواهد بود و در محاسبات تاثیری نخواهد گذاشت را می‌توان حذف کرد و به معادله زیر رسید:

$$p' = p + \dot{p}t$$

و همچنین چنان که دیدیم معدله به روز رسانی سرعت ذره نیز به این صورت خواهد بود:

$$\dot{p}' = \dot{p} + \ddot{p}t$$

اما چنان که قبلا هم اشاره شد ما از damping برای کم کردن مقداری از سرعت در هر به روز رسانی استفاده می‌کنیم. این کار با معادله زیر امکان پذیر است:

$$\dot{p}' = \dot{p}d^t + \ddot{p}t$$

که در آن d همان مقدار damping است که با به توان زمان رساندن آن فاکتور زمان را هم در آن دخیل کرده‌ایم.

به این صورت می‌توانیم Integrator را به صورت زیر پیاده کنیم:

```
Particle.prototype.integrate = function(duration){
   if(duration < 0){
      //exception
      console.log("error in duration");
   }
   this.position.addScaledVector(this.velocity, duration);
   var resultingAcc = new Point(this.acceleration.x,this.acceleration.y);
```

```
    resultingAcc.addScaledVector(this.forceAccum, this.inverseMass);
    this.velocity.addScaledVector(resultingAcc, duration);
    this.velocity.multiplebyScalar(Math.pow(this.damping, duration));

    if(this.bindedShape!=null){
       this.bindedShape.position = this.position;
    }
    this.forceAccum.x=0;
    this.forceAccum.y=0;

    this.path.segments[1].point = this.position;
}
```

۵.۴ جمع‌بندی و نتیجه‌گیری

در این فصل قوانین حرکت را بررسی کردیم و دیدیم که چگونه می توان موقعیت، سرعت و شتاب را در یک کلاس برا ذرات پیاده‌سازی کرد. این پیاده‌سازی‌ها در آخر برای به روز رسانی به Integrator نیاز خواهد داشت که معادلات حرکت را اعمال کند.

در فصل بعد به چند نمونه استفاده از این پیاده‌سازی خواهیم پرداخت.

فصل پنجم: استفاده از فیزیک ذره‌ای در موتور

۱.۵ مقدمه

با مبانی فیزیکی که تا همین جا بحث شده است می‌توان اولین شبیه‌سازی‌های تحت موتور را به وجود آورد که شبیه‌سازی حرکات تحت تاثیر شتاب و سرعت اولیه می‌باشد. نکته اساسی در این است که با این که تا این جا کد بسیار زیادی تولید نشده است اما تئوری فیزیک پشت آن بسیار غنی و مهم می‌باشد. تا این جای کار موتور فیزیک محدودیت‌های بسیاری دارد که در فصل‌های به آن‌ها خواهیم پرداخت برای مثال ذرات تا این جا شیئ‌هایی ایزوله هستند که با محیط خود تعاملی ندارند اما بعدا این تعامل باید ایجاد شود. در این فصل از موتور برای پردازش پرتابه‌ها استفاده می‌کنیم. این پرتابه‌ها شامل گلوله، توپ و غیره می‌شود. همچنین از موتور برای ایجاد آتش‌بازی استفاده خواهیم کرد.

۲.۵ پرتابه‌ها

یکی از بیشترین کاربردهای موتور فیزیک شبیه‌سازی پرتابه‌ها می‌باشد. در مثال این پروژه هر اسلحه یک نوع ذره را شلیک می‌کند. این ذرات که آن‌ها را پرتابه می‌نامیم شامل انواع مختلفی می‌تواند باشد از جمله گلوله تفنگ، توپ جنگی و یا گلوله‌های انفجاری.

هر اسلحه مشخصات دهانه خاص خود را دارد که در نتیجه آن سرعت اولیه خاص خد را به پرتابه می دهد که برای مثال برای یک اسلحه لیزری بسیار سریع و برای یک توپ آتشین کندتر است.

باید توجه داشت که برای انتخاب سرعت اولیه برای پرتابه‌ها آن چه در واقع و در دنیای بیرونی وجود دارد احتمالا در موتور فیزیک کاربرد نخواهد داشت و باید دستکاری شود. برای مثال برای شبیه‌سازی لیزر منطقی نخواهد بود که سرعت آن را ۳۰۰،۰۰۰،۰۰۰ متر در ثانیه در نظر بگیریم چرا که در این صورت، اثری از آن در بازی دیده نخواهد شد و اگر بخواهیم تنها اثر آن را در اجسام دیگر ببینیم منطقی نخواهد بود که اصلا پرتابش کنیم، بلکه می‌توانیم نقطه اثر را مستقیما پیدا کنیم. اما چون در این مثال خود پرتاب شدن لیزر را می

خواهیم مشاهده کنیم می‌توانیم با سرعت کمتری تا جایی که دیده شود اما بالا بودن سرعت آن نیز احساس شود پرتابه مربوط به لیزر را پرتاب کنیم. در ضمن احتمالا لازم است که جرم جسم نیز بیشتر از آنچه در عالم واقع هست در نظر گرفته شود. این موضوع سبب می‌شود که انرژی آن‌ها در هنگام برخورد افزایش پیدا کند و اثر تخریبی بیشتری داشته باشد. در ضمن در مورد شتاب جاذبه نیز باید دستکاری‌هایی انجام دهیم تا برای هر پرتابه سرعت پایین رفتن آن متناسب با نوع پرتابه باشد. یک راه ساده برای تعیین اندازه جاذبه برای هر پرتابه انتخاب آن با توجه به سرعت آن است به این صورت که برای هر تغییر در سرعت پرتابه جاذبه می‌تواند از معادله زیر بدست آید:

$$g_{\text{bullet}} = \frac{1}{\Delta s} g_{\text{normal}}$$

که در آن g نرمال شتابی است که می‌خواهیم شبیه‌سازی کنیم و در دنیای واقع ۱۰ متر بر مجذور ثانیه می‌باشد. در نتیجه برای پرتابه‌ای مثل گلوله این مقدار به ۰.۵ متر بر مجذور ثانیه خواهد رسید.

در کد زیر پیاده‌سازی چهار پرتابه را با کمک این موتور فیزیک که تا این جا طراحی کرده‌ایم آورده شده است. در این کد چهار پرتابه شبیه‌سازی شده اند که گلوله تفنگ، توپ جنگی، توپ آتشین (که بعد از پرتاب آتش گرفته و در صورت عدم برخورد بر اثر جرم کم به هوا می‌رود) و لیزر هستند. برای هر پرتابه سرعت اولیه، شتاب اولیه، مقدار damping، و مقدار جرم آن را مشخص کرده‌ایم. مکان اولیه همه آن‌ها یکسان ور در سمت چپ صفحه است. کاربر با زدن هر عدد از یک تا چهار یکی از آن‌ها را می‌تواند پرتاب کند:

```
var ball;
   tool.onKeyDown = function(event){
      if(event.character=='1'){
         var ball1 = new Particle();
         ball1.position = initialPosition;
         ball1.inverseMass = 1/2;//kg
```

```
        ball1.velocity = new Point(350,0);
        ball1.acceleration = new Point(0,10);
        ball1.damping = 0.99;
        ball1.forceAccum = new Point(0,0);
        ball1.bindedShape = myCircle;
        ball = ball1;
    }else if(event.character=='2'){
        var ball2 = new Particle();
        ball2.position = initialPosition;
        ball2.inverseMass = 1/200;//kg
        ball2.velocity = new Point(400,-300);
        ball2.acceleration = new Point(0,200);
        ball2.damping = 0.99;
        ball2.forceAccum = new Point(0,0);
        ball2.bindedShape = myCircle;
        ball = ball2;
    }else if(event.character=='3'){
        var ball3 = new Particle();
        ball3.position = initialPosition;
        ball3.inverseMass = 1;//kg
        ball3.velocity = new Point(250,0);
        ball3.acceleration = new Point(0,-30);
        ball3.damping = 0.9;
        ball3.forceAccum = new Point(0,0);
        ball3.bindedShape = myCircle;
        ball = ball3;
    }else if(event.character=='4'){
        var ball4 = new Particle();
        ball4.position = initialPosition;
        ball4.inverseMass = 10;//kg
        ball4.velocity = new Point(1000,0);
        ball4.acceleration = new Point(0,0);
        ball4.damping = 0.99;
        ball4.forceAccum = new Point(0,0);
        ball4.bindedShape = myCircle;
        ball = ball4;
    }
}
```

همچنین در مورد به روز رسانی فیزیک هم لازم است کد زیر توجه شود که در هر فریم مدت زمان گذشت آن فریم گرفته می‌شود و به Integrator پرتابه داده می‌شود تا محاسبات لازم را انجام دهد و پرتابه را به روز کند:

```
onFrame = function(event){
    if(ball){
        ball.integrate(event.delta);
    }
}
```

۳.۵ آتش‌بازی

مثال آتش‌بازی نیز می‌تواند توسط موتور فیزیکی که تا این جا طراحی کرده‌ایم مورد پیاده‌سازی قرار بگیرد کاربردهایی که این پیاده‌سازی می‌تواند داشته باشد علاوه بر خود آتش‌بازی انواع انفجارها، پخش شدن آب و حتی دود و آتش است.

برای ایجاد آتش‌بازی باید به نوع Particle چند داده دیگر را نیز اضافه کنیم تا تبدیل به یک شیئ انفجاری شود. اشیا انفجاری اول از همه حاوی بار انفجاری هستند که آن‌ها را payload می‌نامیم. راکت اولیه که پرتاب می‌شود به چندین شیئ انفجاری کوچک دیگر تبدیل می‌شود که آن‌ها هم خود بعد از انفجار می‌توانند بار انفجاری داشته باشند. به این صورت اشیا انفجاری نوع‌های مختلفی دارند که آن‌ها را باید در برنامه در نظر بگیریم.

نکته دوم در آتش‌بازی عمر هر ذره است. باید برای هر ذره انفجاری عمری تعیین شود تا به کمک آن مشخص شود که بعد از مدتی آن ذره منفجر شده و از بین می‌رود، در هنگام انفجار آن نیز بار انفجاری تخلیه می‌شود. این عمر در هر به روز رسانی در هر فریم باید کم شود.

بنابراین برای داشتن این اطلاعات اضافه ما نوع particle را با ارث‌بری به نوع firework گسترش می دهیم:

```
function Firework(){
    Particle.call(this);
    this.type = null;
    this.age = null;
}
```

```
Firework.prototype = new Particle();
Firework.prototype.constructor = Firework;
Firework.prototype.update = function(duration)
{
   this.integrate(duration);
   this.age -= duration;
   return (this.age < 0);
}
```

این موضوع که هر نوعی از شیئ انفجاری چه خصوصیاتی دارد و مثلا بار انفجاری دارد و اگر دارد آن باز ها چقدر و هر کدام از چه نوعی هستند را نیز باید با ساختاری مناسب در کد بگنجانیم برای این کار ساختار FireworkRule را در نظر گرفته‌ایم تمام این اطلاعات را در بر می‌گیرد.

```
function FireworkRule()
{
   this.type = null;
   this.minAge = null;
   this.maxAge = null;
   this.minVelocity = null;//vector
   this.maxVelocity = null;//vector
   this.damping = null;

   this.payloadCount = null;
   this.payloads = new Array();
}
function Payload(type,count)
{
   this.type = type;
   this.count = count;
}
```

قوانین در کد تعریف می‌شوند و لیستی از آن‌ها به وجود می‌آید و لازم به توضیح جزئیات آن‌ها در این جا نمی‌باشد.

در پیاده‌سازی هر شیئ انفجاری عمرش به روز می‌شود و با قوانین مربوط به خود مقایسه می‌شود. اگر عمرش تمام شود از ازی حذف شده و اشیا انفجاری دیگری به جایش با توجه به قانون متناظرش ایجاد می‌شود. کدی که اشیا انفجاری جدید را می‌سازد به این صورت است:

```
FireworkRule.prototype.create = function(parent)
{
var firework = new Firework();
   firework.type = this.type;
   firework.age = (this.maxAge-this.minAge)*Math.random()+this.minAge;
   if (parent!=null){
      var position = new Point();
      position.x = parent.position.x;
      position.y = parent.position.y;
      firework.position = position;
      var vel = new Point();
      vel.x = parent.velocity.x;
      vel.y = parent.velocity.y;
      vel.x += (this.maxVelocity.x-this.minVelocity.x)*Math.random()+this.minVelocity.x;
      vel.y += (this.maxVelocity.y-this.minVelocity.y)*Math.random()+this.minVelocity.y;
      firework.velocity = vel;
   } else{
      firework.position = new Point(300*Math.random()+200,650);
      var vel2 = new Point(0,0);
      vel2.x += (this.maxVelocity.x-this.minVelocity.x)*Math.random()+this.minVelocity.x;
      vel2.y += (this.maxVelocity.y-this.minVelocity.y)*Math.random()+this.minVelocity.y;
      firework.velocity = vel2;
   }
   firework.inverseMass = 1;
   firework.damping = this.damping;
   firework.acceleration = new Point(0,50);
   return firework;
}
```

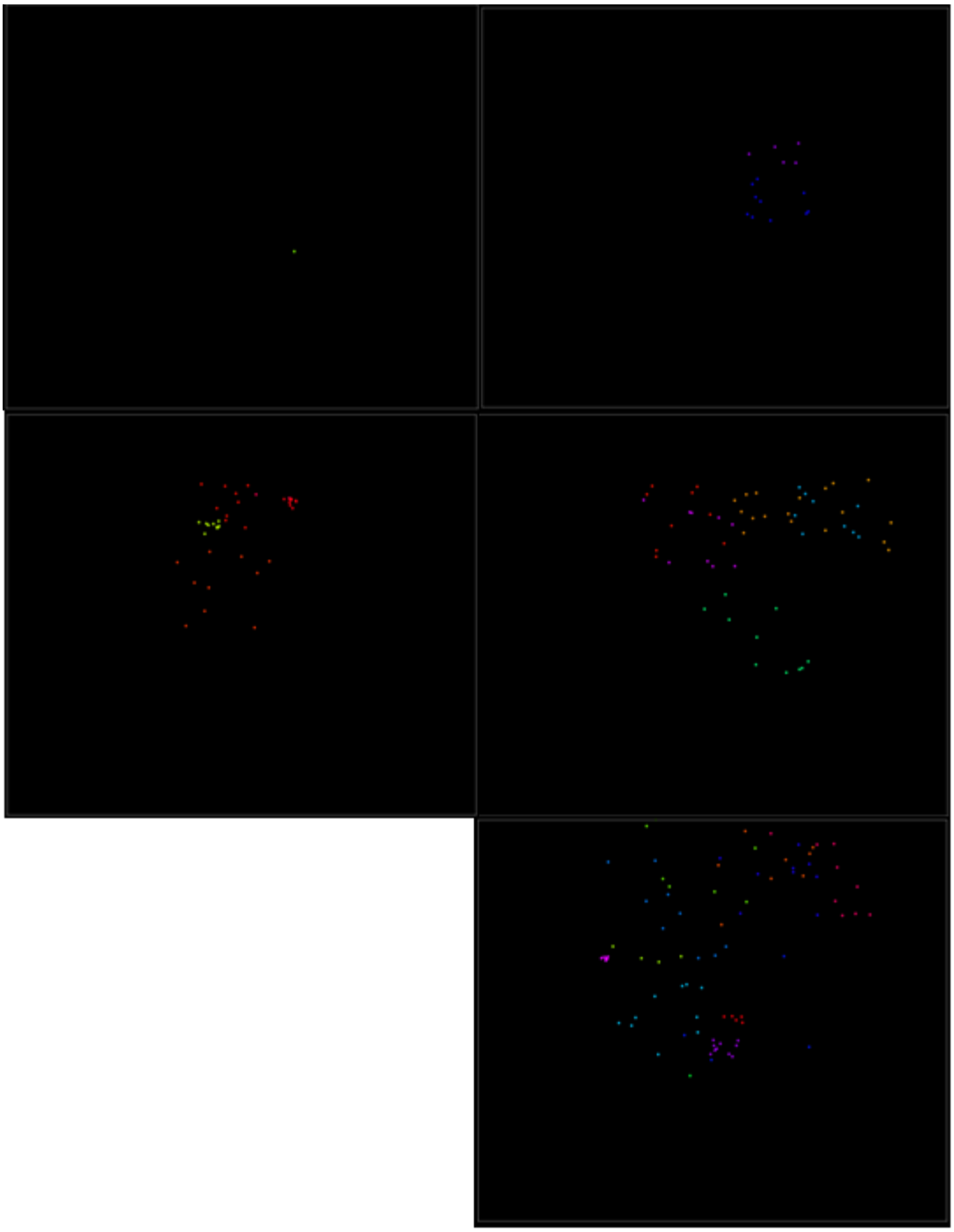
وقتی اشیا ایجاد می‌شوند مقدار متغیرهایشان با توجه به متغیرهای شیئی که در ساختشان نقش داشته و قوانین مربوط به خودشان مقداردهی می‌شود. Damping آن‌ها با توجه به میزان مسافتی که لازم است بپیمایند تعیین می‌شود و جاذبه نیز بر آن‌ها تاثیرگذار است.

در هر فریم تمام اشیا موجود به روز می‌شوند و اگر هر کدام تمام شود اقدام لازم انجام می‌شود.

```
for(var i =0 ; i< fireworks.length;i++){
        var firework = fireworks[i];
        if (firework.type > 0)
        {
           if (firework.update(event.delta))
```

```
        {
            var rule = rules[firework.type-1];
                                    firework.type = 0;
            firework.bindedShape.remove();
            var index = fireworks.indexOf(firework);
            fireworks.splice(index, 1);
            for (var j = 0; j < rule.payloadCount; j++)
            {
                var payload = rule.payloads[j];
                create(payload.type, payload.count, firework);
            }
        }
    }
```

در شکل زیر نمونه‌هایی از لحظاتی از اجرای این مثال آورده شده است.



شکل ۴ مثال آتش بازی

۴.۵ جمع‌بندی و نتیجه‌گیری

بنابراین همان طور که مشاهده کردیم موتور فیزیکی که تا این جا طراحی و ایجاد کرده‌ایم می‌تواند برای ایجاد جلوه‌های ویژه به کار رود. حرکت پرتابی پرتابه‌ها، سیتم‌های ذره‌ها و جلوه‌های ویژه انفجارها از این جمله هستند. با انجام تنظیمات فراهم شده برای ذرات چون جاذبه، مقاومت محیط و سرعت اولیه می‌توان اکنون انواع پدیده‌های فیزیکی چون حتی آبا سیال و دود و آتش‌بازی را شبیه‌سازی کرد. در قسمت‌های بعد گسترش این موتور فیزیک را بررسی می‌کنیم تا انواع دیگر پدیده‌های فیزیکی و اشیا را بتوانیم شبیه‌سازی کنیم.

فصل ششم: فیزیک تجمیع اجرام، نیروهای عمومی

۱.۶ مقدمه

در طراحی موتور فیزیک تا این جا نیروهایی چون جاذبه زمین را با مشخص کردن شتاب آن‌ها در شبیه سازی گنجانده‌ایم اما لازم است تا موتور را برای این که با هر تعداد نیرو که به هر تعداد ذره وارد می‌شود عمل کند گسترش دهیم. جاذبه را نیرویی جداگانه در نظر بگیریم و تولید کننده‌های نیرو را بررسی کنیم.

۲.۶ اصل دالامبر[۲]

ما معادله‌های لازم برای این که رفتار شیئ تحت تاثیر یک نیرو را تحلیل کنیم در اختیار داریم اما اگر چند نیرو همزمان بر یک شیئ تاثیر بگذارند چگونه باید رفتار شیئ را مشخص کنیم؟ نیروهای مختلفی که بر روی ذره تاثیر می‌گذارن هر کدام اثر متفاوتی بر روی آن خواهند گذاشت و جمع این آثار برای تحلیل در موتور بسیار اهمیت دارد. اما اگر بتوان تاثیر تمام نیروها را یکجا بررسی کرد کمک بزرگی در پیاده‌سازی موتور خواهد بود.

اصل دالامبر در این جا به کمک می‌آید. هر چند تمام اصل در واقع پیچیده و برای موتور بدون کاربرد است و در واقع کمیت‌های در ایجاد معادله‌های حرکت را به هم مربوط می‌کند. اما یک کاربرد بسیار مهم از این اصل را می‌توان در این جا بیان و از آن استفاده کرد.

این کاربرد به این قرار است که اگر یک مجموعه از نیروها بر روی یک شیئ تاثیر بگذارند می‌توان تمام آن نیروها را با یک نیرو جایگزین کرد که به این صورت محاسبه می‌شود:

$$f = \sum_i f_i$$

[۲] D'ALEMBERT

به عبارت دیگر به سادگی می‌توانیم تمام نیروهای وارده را با جمع برداری با هم جمع کرده و یک نیروی واحد بدست آوریم و سپس اثر آن را به عنوان اثر کل نیروها به کار بگیریم.

برای این کار یک بردار جمع نیروها در particle در نظر می‌گیریم که در هر بار به روز رسانی آن را صفر کرده و نیروهای وارده را سپس در آن جمع می‌کنیم. نیروی جمع شده آخر را سپس در محاسبات وارد می‌کنیم:

```
this.forceAccum = new Point(0,0);
```

برای این که نیروها را به ذرات بتوانیم وارد کنیم می‌شود آن‌ها را در هر بار به روز رسانی در forceAccume جمع کرد تا در ذره اثر بگذارند امااین تنها برای چند نیروی محدود و مدت زمان محدود امکان پذیر است. باید روشی را اتخاذ کنیم تا برای هر چند نیرو و برای هر چند ذره و هر چقدر زمان بتوانیم این کار را انجام دهیم. برای همین از یک registry برای ثبت نیروها و ذره اثر پذیرشان استفاده می‌کنیم. در یک رجیستری هر نیرو با یک ذره ثبت می‌شود که باید برای آن در هر فریم نیرو را تامین کند. این مولدهای نیرو را در برنامه force generator می‌نامیم.

۳.۶ تامین‌کنندگان نیرو

در این موتور فیزیک مکانیزم تاثیر چندین نیرو با هم را ایجاد کرده‌ایم اما این که نیروها چگونه و از کجا به وجود می‌آیند را نیز باید دقیقا مشخص کنیم. نیرویی مانند جاذبه در این مورد ساده به نظر می‌رسند، چون همیشه برای همه ذرات وجود دارد. اما نیروهای دیگر لزوما به این شکل نیستند.

بعضی نیروها به دلیل رفتار شیئ به وجود می‌آیند. برای مثال نیروی مقاومت محیط برای هر ذره با سرعت‌های متفاوت متفاوت است. نیروهایی کاملا از محیطی می‌آیند که ذره در آن‌ها هست. مثل نیروی رانش آب به ذره شناور. بعضی نیروها در ارتباط بین ذرات با هم به وجود می‌آیند مانند نیروی ناشی از فنر که در فصل

بعد بررسی می‌شود. و در آخر نیروهای دیگری هستند که از درخواست کاربر چه حقیقی و چه کنترل شده توسط هوش مصنوعی به وجود می‌آیند به مانند درخواست افزایش شتاب ماشین.

پیچیدگی دیگر نیروها در طبیعت پویای بعضی از آن‌هاست. نیرویی مثل نیروی جاذبه زمین از نظر ساده است چون همیشه و همه جا ثابت است. می‌توان آن را تنها یک بار محاسبه کرد و در تمام بازی به کارش برد. اما اکثر نیروهای دیگر دائما در حال تغییرند. بعضی با توجه به تغییر موقعیت ذره و یا سرعت آن تغییر می کنند: نیروی مقاومت محیط در سرعت‌های بالاتر بیشتر است و نیروی ناشی از فنر در فشرگی‌های بیشتر، باز بیشتر است. بعضی دیگر اما با توجه به عوامل خارجی تغییر می‌کنند: یک انفجار یا تمام شدن سوخت یک محرک.

ما باید قادر باشیم تا با انواع مختلفی از نیروها که با انواع متفاوتی از مکانیزم‌ها برای محاسبات شان وجود دارند کار کنیم و آن را در موتور خود داشته باشیم. بعضی ثابت‌اند، بعضی توابعی را روی برخی خصوصیات ذره‌ها اعمال می‌کنند، بعضی ورودی از کاربر می‌خواهند و بعضی وابسته به زمانند. اگر تمام این انواع را بخواهیم در کد پیاده‌سازی کنیم و از کاربر انتظار داشته باشیم که برای انواع نیروهایی که در نظر دارد آن‌ها را با هم ترکیب و از بینشان انتخاب کند بعد از چند نوع پیاده‌سازی به کدی خواهیم رسید که از نظر مدیریتی بسیار سخت و پیچیده خواهد شد. به طور ایده‌آل اما در نظر داریم که نحوه محاسبه و تولید نیرو را بتوانیم از بقیه قسمت‌ها ایزوله کنیم و به موتور این اجازه را بدهیم که فارغ از این که نیرو چگونه تولید می‌شود با آن بتواند کار کند. به این صورت می‌توانیم هر تعداد نیرو که بخواهیم به یک شیئ وارد کنیم بدون این که نیاز باشد که شیئ از نحوه تولید نیروها با خبر شود.

این کار با ساختاری به نام force generator در این موتور امکان پذیر است. در واقع به تعداد انواع نیروها می‌تواند مولد نیرو وجود داشته باشد همان شیئ از درون آن‌ها و نوع آن‌ها نیاز ندارد که باخبر باشد. ذرات

از یک رابط ثابت برای یافتن نیرویی که به آن‌ها وارد می‌شود استفاده می‌کنند و این نیروها در قسمت integrator با هم جمع شده و اعمال می‌شوند. به این صورت می‌توانیم هر تعداد نیرو با هر نوعی را به هر ذره‌ای وارد کنیم. و درضمن می‌توانیم برای هر بازی و یا هر مرحله این نوع‌های جدیدی از نیرو را به راحتی ایجاد کنیم بدون این که نیاز باشد تا تغییرات عمده‌ای در موتور انجام دهیم و کدی را بازنویسی کنیم.

برای رسیدن به این مقصود نیاز داریم تا از یک مفهوم شیئ‌گرایی در JavaScript استفاده کنیم که در prototype-based بودن این زبان نهفته است و آن توانایی بسیار بالای آن در ایجاد polymorphism و استفاده از pseudo-interfaceها می‌باشد. در زبان‌های شیئ‌گرا interface در واقع مشخصات کارکردی یک نوع را بیان می‌کند و مشخص می‌کند که یک نوع شیئ چگونه با نوع‌های دیگر ارتباط برقرار می‌کند و در واقع توابعی از آن نوع که دیگران می‌توانند استفاده کنند را در معرض می‌گذارد. شیئ‌هایی که این قرارداد را رعایت کنند درواقع آن interface را پیاده کرده‌اند. قدرت interfaceها در واقع در polymorphism نهفته است که توانایی یک زبان برای استفاده از قطعه‌های نرم‌افزاری است که آن interface را پیاده کرده‌اند. در واقع کد استفاده کننده از یک interface می‌تواند از هر شیئی که آن را پیاده کرده باشد با توجه به توابهی که در آن توضیح داده شده است استفاده کند. کد صدا زننده هرگز از پیاده‌سازی درونی خبر ندارد چنان که برای هر شیئ هم متفاوت است. جانشین‌پذیری در واقع نکته اساسی این خاصیت است چنان که ما یک interface برای مولدهای نیرو داریم که می‌توانیم از آن‌ها برای تامین نیرو استفاده کنیم، حال هر نوع مولد نیرو را می‌توانیم ایجاد کنیم و ذره‌ها بدون دانستن نوع آن‌ها می‌توانند از آن‌ها استفاده کنند.

در پیاده‌سازی مولدهای نیرو، آن‌ها تنها نیروی زمان حال را ارائه می‌کنند که در نیروهای دیگر جمع شده و به ذره اعمال می‌شود:

```
function ParticleForceGenerator(){
    }
```

```
ParticleForceGenerator.prototype.updateForce = function(particle,duration){
}
```

متد updateForce با زمان فریمی‌که نیرو را در آن می‌خواهیم و خود ذره فراخوانی می‌شود. این زمان در برخی از تامین کنندگان نیرو ضروری است، مانند تامین‌کنندگان نیروی فنری که به شدت به این زمان وابسته‌اند.

در ضمن خود particle را هر بار به این متد می‌دهیم تا نیاز نباشد خود مولد نیرو از مشخصات آن جداگانه نگهداری کند. به این صورت یک نمونه از یک مولد می‌تواند برای چندین ذره به کار رود و برای هر کدام که تابعش فراخوانی شود کار محاسبات را انجام دهد. همچنین مولد نیرو مقداری را باز نمی‌گرداند، به این صورت انعطاف‌پذیری آن حفظ شده و برای به روز کردن نیروی ذره مستقیما خود عمل می‌کند تا انواع عملیت دلخواه را بتواند انجام دهد به جای این که مقداری را بازگرداند.

همچنین لازم است که ساختارهایی را برای حفظ تامین‌کنندگان و ذرات نظیرشان در نظر بگیریم. برای رسیدن به بهترین کارایی اجرایی و حافظه یک ساختار ثبت کلی و مرکزی برای همه ذرات و تامین کنندگان به وجود می‌آوریم.

```
function ParticleForceRegistry(){
   this.registrations = new Array();
}
function ParticleForceRegistration(){
   this.particle = null;
   this.particleForceGenerator = null;
}
```

در هر فریم قبل از این که به روز رسانی انجام شود تمام تامین کنندگان نیرو صدا زده می‌شوند و نیروهای خود را به ذرات اضافه می‌کنند تا بعدا در به روز رسانی به شتاب مناسب برسند:

```
ParticleForceRegistry.prototype.updateForces = function(duration){
```

```
    for(var i=0;i<this.registrations.length;i++){
        this.registrations[i].particleForceGenerator.updateForce(this.registrations[i].particle,
duration)
    }
}
```

با توجه به پیاده‌سازی ساختاری که تاکنون داشته‌ایم اکنون می‌توانیم یک مولد نیرو برای جاذبه زمین ایجاد کنیم و به جای روش قبلی که یک شتاب ثابت را به ذره وارد می‌کرد قرار دهیم. این پیاده‌سازی به این شکل خواهد بود:

```
function ParticleGravity(){
    ParticleForceGenerator.call(this);
    this.gravity = new Point(0,10);
}
ParticleGravity.prototype = new ParticleForceGenerator();
ParticleGravity.prototype.constructor = ParticleGravity;
ParticleGravity.prototype.updateForce = function(particle,duration){
    if(!particle.infiniteMass){
        particle.addForce(new
Point((1/particle.inverseMass)*this.gravity.x,(1/particle.inverseMass)*this.gravity.y));
    }
}
```

نیرو با نجوه به جرم جسمی‌که به تابع داده شده است محاسبه می‌شود، تنها داده‌ای که توسط اسن شیئ نگهداری می‌شود شتاب گرانش است و یک نمونه از این شیئ می‌تواند به هر تعداد ذره اعمال شود.

اکنون می‌توانیم نیروهای دیگری را نیز پیاده‌سازی کنیم. نیروی بعدی، نیروی مقاومت محیطی است. این نیرو، نیرویی است که به بدنه شیئ وارد می‌شود و به سرعت آن وابستگی دارد. معادلات نیروی مقاومت محیطی معادلات پیچیده‌ای هستند که در زمان واقعی قابل پیاده‌سازی نمی‌باشند. اما در موتور فیزیک یک نوع ساده‌تر از این معادلات را به کار می‌بریم که از معادله زیر بدست می‌آید:

$$f_{\text{drag}} = -\hat{\dot{p}}\left(k_1|\hat{\dot{p}}| + k_2|\hat{\dot{p}}|^2\right)$$

که در آنها دو ضریب k ثابت‌هایی هستند که مشخص می‌کنند که نیروی مقاومت چقدر قوی است. این ثابت‌ها ضریب‌های مقاومت محیطی نامیده می‌شوند که هم به نوع شیئ و هم نوع محیط بستگی دارند.

این معادله بیان می‌کند که این نیرو در جهت مخالف سرعت ذره وارد می‌شود با قدرتی که به مقدار سرعت و مجذور این مقدار رابطه مستقیم دارد.

نیروی مقاومتی که ضریب دوم را دارد سرعت بسیار بالاتری رشد می‌کند. این همان دلیلی است که یک خودرو نمی‌تواند از یک سرعت بیشتر حرکت کند والا به سرعت بی نهایت هم می‌رسید. در سرعت‌های کم خودرو عملا هیچ نیرویی را از محیط حس نمی‌کند ولی برای هر دو برابر شدن سرعتش، این نیرو چهار برابر می شود.

پیاده‌سازی این نیرو به شکل زیر می‌باشد:

```
function ParticleDrag(){
   ParticleForceGenerator.call(this);
   this.k1 = 1;
   this.k2 = 0;
}
ParticleDrag.prototype = new ParticleForceGenerator();
ParticleDrag.prototype.constructor = ParticleDrag;
ParticleDrag.prototype.updateForce = function(particle,duration){
   var force;
   force = particle.velocity;
   var dragCoeff = force.length;
   dragCoeff = this.k1 * dragCoeff + this.k2 * dragCoeff * dragCoeff;
   force = force.normalize();
   force.x *= -dragCoeff;
   force.y *= -dragCoeff;
   particle.addForce(force);
}
```

در این جا نیز نیروی ایجاد شده به خصوصیات ذره بستگی دارد و تنها دو ثابت ضریب گفته شده در مولد نگهداری می‌شوند و به مانند جاذبه این مولد می‌تواند یک نمونه اش برای چندین ذره به کار رود. این مدل

مقاومت محیطی از آن damping که در خود particleها به کار می‌رود بسیار پیچیده‌تر است و در موارد با دقت بالا و با حس واقعی‌تر می‌تواند به کار رود.

۴.۶ جمع‌بندی و نتیجه‌گیری

دیدیم که چگونه نیروهای مختلفی با جمع شدن می‌توانند عمل نهایی شان را با هم به ذره وارد کنند. این نتیجه اصل دالامبر بوده که به ما این امکان را داد تا هر تعداد نیرو را بر هر تعداد ذره بدن دانستن این که نیرو چگونه به وجود می‌آید اعمال کنیم. در فصل‌های بعد تامین کنندگان نیروی دیگری و پیاده می‌کنیم که در همین ساختاری که این جا پیاده کردیم می‌گنجند.

در تصویر زیر یک نمایش از آنچه در این فصل توضیح داده شد را می‌بینیم. در این مثال به تمام ذرات یک مولد نیروی جاذبه ثابت اعمال کرده‌ایم اما به هرکدام نیروی مقاومت هوای متفاوتی با ضریب مقاوت اول متفاوت و ضریب مقاوت دوم صفر داده‌ایم. همان طور که مشاهده می‌کنیم این ذرات با آن که از نقطه ثابتی رها می‌شوند اما به دلیل نیروی مقاومت هوای متفاوتی که بر آن‌ها حاکم است، بعد از مدتی سرعتشان به طور متفاوت کاسته می‌شود تا این که برخی با این که نیروی جاذبه شتاب می‌خواهد در آن‌ها ایجاد کند اما به دلیل نیروی مقاومت زیاد آن نیرو خنثی شده و به سرعت ثابت می‌رسند.

شکل۵ مثال مولدهای نیرو

فصل هفتم: فنرها و خواص فنری

۱.۷ مقدمه

یکی از کارامدترین نیروهایی که در موتور فیزیک می‌توانیم پیاده کنیم، نیروی فنری است. فنرها برای نشان دادن اشیا نرم و قابل تغییر شکل دادن به کار می‌روند. همچنین فنرها و ذرات می‌توانند برای ایجاد طیف وسیعی از شکل‌ها و پدیده‌های فیزیکی به کار روند که از آن جمله می‌توان طناب‌ها، پرچمها و حتی موج دریا را ذکر کرد.

برای گنجاندن فنرها در این موتور فیزیک ابتدا تئوری فنرها مرور می‌شود و سپس بررسی می‌شود که چطور این تئوری می‌تواند در موتور پیاده شود.

۲.۷ قانون هوک

قانون هوک ریاضیات فنرها را در اختیار ما می‌گذارد. این قانون بیان می‌کند که نیروی ناشی از فنر بر جسم متصل به انتهای آن تنها ناشی از فاصله جسم از جایی است که فنر در حالت استراحت است. این فاصله می‌تواند فاصله فشرده شده و یا گسترش یافته فنر باشد. فنری که دو برابر از دیگری کشیده شده باشد دو برابر بیشتر نیرو اعمال می‌کند. معادله این قانون به این صورت است:

$$f = -k\Delta l$$

که در آن دلتای L نشان دهنده میزان اختلاف طول فنر از طول حالت عادی آن است و k ثابت فنر است که مقداری است که سختی فنر را تعیین می‌کند. این نیرو در دو طرف فنر وارد می‌شود. به عبارت دیگر اگر دو جسم در دو سوی فنر قرار گیرند به هر دو یک مقدار نیرو و با توجه به این معادله وارد می‌شود. با توجه به این موضوع که در طول نرمال و استراحت فنر هیچ نیرویی وارد نمی‌شود این معادله را می‌توان به شکل زیر نوشت:

$$f = -k(l - l_0)$$

اما این معادله مربوط به زمانی است که در یک بعد فنر را بررسی می‌کنیم، اگر بخواهیم آن را در دو بعد و با بردارها در نظر بگیریم این معادله به شکل زیر در خواهد آمد:

$$\boldsymbol{f} = -k(|\boldsymbol{d}| - l_0)\hat{\boldsymbol{d}}$$

که در آن d از معادله زیر بدست می‌آید:

$$\boldsymbol{d} = x_A - x_B$$

که در آن‌ها مختصات نقاط ابتدا و انتهای فنرها داده شده است. این معادله نشان می‌دهد که نیرو باید در جهت انتهای دیگر فنر به جسم اعمال شود و با بزرگی ضریب فنر در مقدار خارج شدن آن از حالت نرمال. در واقع همین معادله را می‌توان برای طرف دیگر فنر نیز به کار برد و نیروی وارده بر طرف دیگر را حساب کرد. چون این محاسبات یک بار تنها برای یک طرف صورت می‌گیرد با دانستن طرف دیگر می‌توان آن را برای آن هم به کار برد.

تنها فنرها اما نیستند که خاصیت فنری دارند و می‌توانند از معادله هوک استفاده کنند. قانون هوک در تعداد زیادی از پدیده‌های طبیعی عمل می‌کند. در واقع هر چیزی که خاصیت ارتجاعی داشته باشد دارای محدودیت ارتجاعی هستند که در آن محدوده قانون هوک عمل می‌کند. کاربردهایی از این دست بسیاز زیادند. طناب‌های بانجی می‌توانند با استفاده از این قانون پیاده شوند. نیروی ناشی از مایعات نیز با اتصال قسمت زیر آب رفته جسم با سطح آب توسط یک فنر فرضی می‌تواند پیاده شود.

۳.۷ مولدهای نیروی فنرگونه

در این جا به بررسی پیاده‌سازی چند نوع مولد نیرو که با قانون هوک کار می‌کنند می‌پردازیم که تفاوت عمده آن‌ها با هم نحوه محاسبه طول فعلی در قانون است.

این قسمت در واقع یک ویژگی بسیار مهم از سیستم‌های فیزیکی را نشان می‌دهد که آن یک هسته اصلی پردازش را به صورت عمومی‌انجام می‌دهد ولی توسط کلاس‌ها و توابع کمکی احاطه شده که بسیار نزدیک به هم می‌باشند و در این جا همان انواع تامین‌کنندگان نیروی فنرگونه می‌باشند.

مولد نیرو فنری پایه طول فنر را به سادگی محاسبه کرده و با قانون هوک نیروی ناشی از آن را استخراج می‌کند و به این صورت پیاده می‌شود:

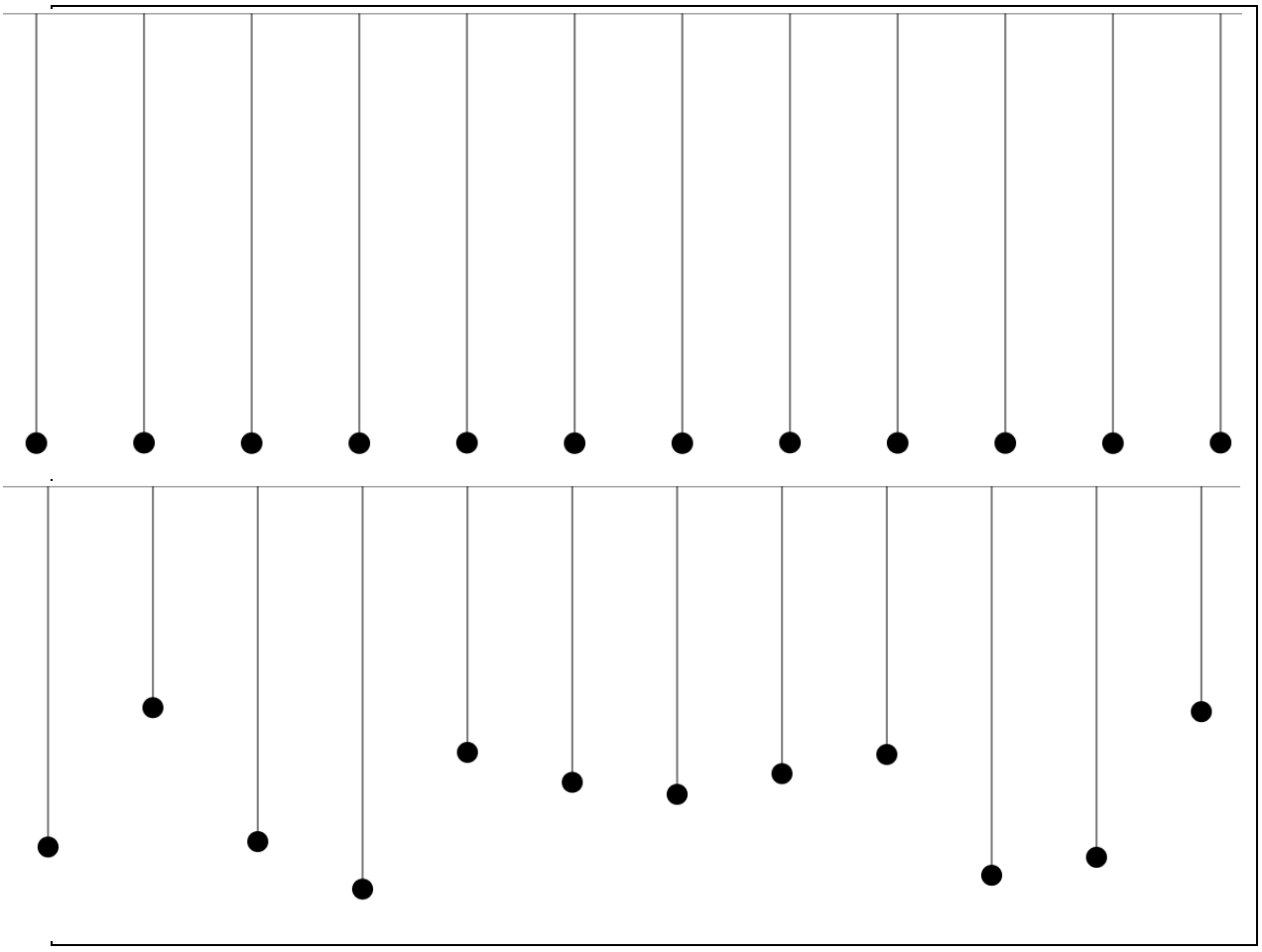
```
function ParticleSpring(other,springConstant,restLength){
   ParticleForceGenerator.call(this);
   this.otherParticle = other;
   this.springConstant = springConstant;
   this.restLength = restLength;
}
ParticleSpring.prototype = new ParticleForceGenerator();
ParticleSpring.prototype.constructor = ParticleSpring;

ParticleSpring.prototype.updateForce = function(particle,duration){
   var force = new Point(particle.position.x,particle.position.y);
   force.x -= this.otherParticle.position.x;
   force.y -= this.otherParticle.position.y;
   var magnitude = force.length;
   magnitude = (magnitude - this.restLength);
   magnitude *= this.springConstant;
   force = force.normalize();
   force.x *= -magnitude;
   force.y *= -magnitude;
   console.log(force.y);
   particle.addForce(force);
}
```

مولد با سه پارامتر ساخته می‌شود. یکی ذره در انتهای دیگر است، دوم ثابت فنر و سوم طول استراحت فنر است.

این مولد چون برای هر فنر متفاوت است یک نسخه اش نمی‌تواند برای هر ذره‌ای به کار رود و تنها برای یک ذره می‌تواند به کار رود. همچنین این که برای دوسوی فنر لازم است که دو مولد جدا اما مثل هم ایجاد و به آن‌ها اعمال شود.

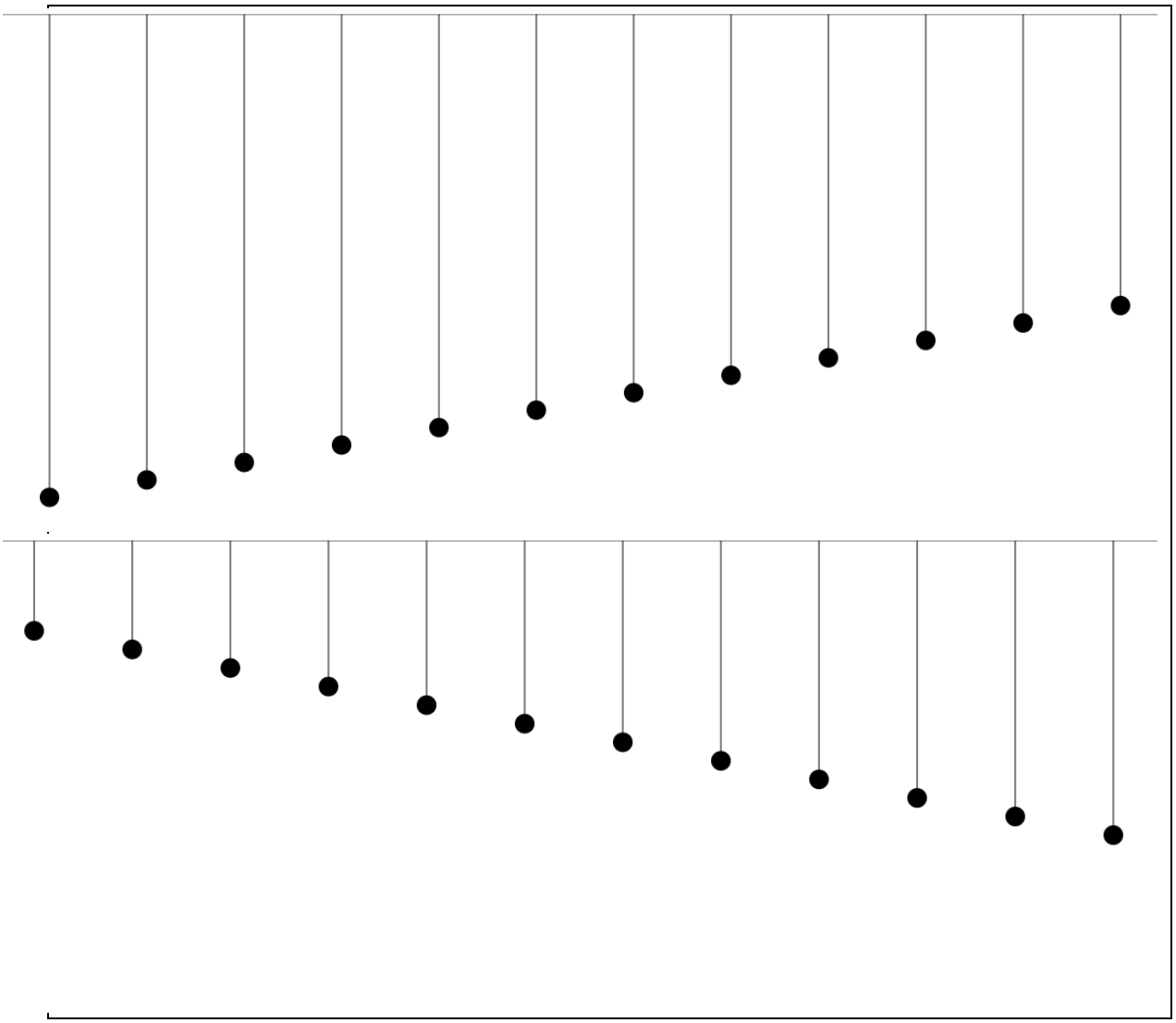
در بسیاری از مواقع نیازی به وجود دو جسم در دو طرف فنر وجود ندارد و نیاز هست که یک طرف فنر به جایی ثابت متصل باشد. در این صورت این نوع مولد نیرو که تا این جا درباره آن صحبت کرده‌ایم به کار نخواهد آمد و با تغییری جزئی یک نوع دیگر که این خاصیت را داشته باشد می‌توانیم ایجاد کنیم. کد به گونه‌ای تغییر می‌یابد تا به جای اتصال آن به یک شی دیگر به یک موقعیت برداری متصل شود و طول‌ها از آن محاسبه شوند.

در شکل زیر یک مثال استفاده از مولد نیروی فنر را مشاهده می کنید. در این مثال به چندین فنر که هر کدام یک ثابت قانون هوک متفاوت دارند و یک انتهای آن‌ها به بالای صفحه متصل شده است یک جسم‌هایی با جرم‌های یکسان متصل شده‌اند. همگی فنرها دارای طول استراحت یکسان و در لحظه ابتدایی حرکت از یک طول باز هم یکسان رها می شوند. اما همان طور که در تصویر دوم مشخص است داشتن ثابت‌های فنری متفاوت برای فنرها با آنکه تمام شرایط دیگر یکسان است باعث می شود که رفتار متفاوت و متناسب با ثابت خود نشان دهند:



شکل ۶ فنرها

در یک مثال دیگر که تصویر آن در پایین آمده است می توان مشاهده کرد که چگونه کشش متفاوت یک نوع فنر در چندین شکل مختلف به رفتار متناسب و در نسبت با مقدار انحراف از طول استراحت فنر می انجامد. همچنین یک نکته پنهان در مثال زیر نگهداری انرژی جنبشی در فنر است که با توجه به تغییر مقدار سرعت با توجه به موقعیت جسم مشاهده می شود:



شکل ۷ مثال دوم فنرها

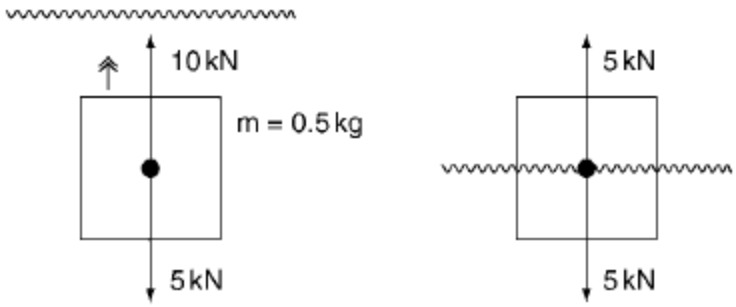
یک طناب بانجی ارتجاعی می‌تواند نیروی کششی ایجاد کند، می‌توان آن را فشرده کرد اما نیروی پرت کننده‌ای ایجاد نخواهد کرد اما مانند هر فنر دیگری وقتی کشیده می‌شود نیروی کششی ایجاد می‌کند. این خاصیت برای نگهداری دو جسم در فاصله‌ای از هم مفید است. دو جسم اگر از هم بیش از حد دور شوند به صرف هم کشیده می‌شوند اما هر چقدر که بخواهند می‌توانند به هم نزدیک شوند بدون این که از هم جدا شوند. این مولد به این شکل پیاده می‌شود:

```
function ParticleBungee(other,springConstant,restLength){
```

```
    ParticleForceGenerator.call(this);
    this.otherParticle = other;
    this.springConstant = springConstant;
    this.restLength = restLength;
}
ParticleBungee.prototype = new ParticleForceGenerator();
ParticleBungee.prototype.constructor = ParticleSpring;
ParticleBungee.prototype.updateForce = function(particle,duration){
    var force = new Point(particle.position.x,particle.position.y);
    force.x -= this.otherParticle.position.x;
    force.y -= this.otherParticle.position.y;
    var magnitude = force.length;
    if(magnitude<=this.restLength){
        return;
    }
    magnitude = Math.abs(magnitude - this.restLength);
    magnitude *= this.springConstant;
    force = force.normalize();
    force.x *= -magnitude;
    force.y *= -magnitude;
    particle.addForce(force);
}
```

همان طور که برای فنرها نیز یک نسخه تک جسمی‌ایجاد کردیم برای bungee هم می‌توانیم این مولد نیرو را بدون نیاز به جسم دیگر ایجاد کرده و طرف دیگر را به یک موقعیت در صفحه متصل کنیم که ثابت می باشد.

نیروی شناوری نیرویی است که باعث شناور ماندن جسم در مایع می‌شود. ارشمیدس برای اولین بار بیان کرد نیروی شناوری با مقدار مایعی که جسم با داخل شدن به مایع جابجا می‌کند نسبت مستقسم دارد. در شکل زیر مشاهده می‌کنیم که یک جسم به جرم ۰.۵ کیلوگرم در آب فرو رفته است.ای خالط چگالی ۱۰۰۰ کیلوگرم بر متر مکعب را دارد، یعنی یک متر مکعب از آب جرم یک تن را دارد. جسم شکل زیر ۰.۰۰۱ متر مکعب حجم دارد، بنابراین همین مقدار آب را جابجا می‌کند. بنابراین این مقدار آب یک کیلوگرم جرم خواهد داشت.



شکل ۸ نیروهای شناوری

باید توجه داشت که جرم به معنی وزن نیست و وزن نیرویی است که به واسطه جاذبه بر جرم ایجاد شده است و با توجه به تغییر جاذبه می‌تواند تغییر کند هر چند که جرم همیشه ثابت است و از معادله زیر بدست می‌آید:

$$f = mg$$

بر روی زمین مقدار g ۱۰ فرض می‌شود. بنابراین جسم مورد نظر در شکل سمت چپ که کاملا در آب فرور رفته است نیروی kN ۱۰ را از طرف مایع به خود می‌بینند. اما جسم سمت راست که به طور نسبی در آب فرو رفته است تنها kN ۵ نیرو را از طرف آب دریافت می‌کند. وزن جسم در هر دو شکل یکسان است و چون در سمت چپ کمتر از نیروی شناوری است به سمت بالا حرکت می‌کند اما وقتی به وضعیت شکل سمت راست می رسد نیروها با هم برابر شده و جسم حالت ایستایی پیدا می‌کند و شناور می‌ماند.

محاسبه مقدار دقیق نیروی شناوری بستگی به دانش در رابطه با شکل جسم دارد، چرا که این شکل جسم است که تعیین می‌کند چقدر از مایع جابجا شده است که در محاسبه نیروی ناشی از آن نقش داشته باشد. اما در واقع آن چنان که به واقع نیاز هست، این مقدار جزئیات عملا در موتور فیزیکی لازم نیست، با محاسبه تقریبی شکل جسم به جسم‌های ساده هندسی و به خصوص مستطیل‌های دوری هر شکلی می‌توان به حس نیروی شناوری نسبتا دقیقی رسید.

در نتیجه می‌توانیم از یک تقریب فنرگونه برای محاسبه این نیرو استفاده کنیم. وقتی شیئ در نزدیکی سطح آب است از یک نیروی مانند فنر استفاده می‌کنیم تا نیروی شناوری را بتوانیم ایجاد کنیم. این نیرو با میزان عمق فرو رفتگی جسم در مایع نسبت مستقیم دارد، همان طور که در فنر نیروی حاصله با مقدار کشیدگی و فشردگی از اندازه نرمال نسبت مستقیم دارد. همان طور که در شکل هم دیدیم این روش برای شیئ مستطیلی که کاملا در آب فرو نرفته است دقیق است. برای هر نوع شیئ دیگر اما یک تقریب خوبی است که هر چند دقیق نیست.

وقتی جسم کاملا در آب فرو رفته باشد، رفتار متعاقب با آن کمی‌متفاوت خواهد بود. در این حالت عمیق‌تر رفتن آن به داخل آب باعث نخواهد شد که آب بیشتری جابجا شود، بنابراین از آن جایی که آب چگالی یکسانی خواهد داشت این کار نیروی وارده را افزایش نخواهد داد. اما یک نکته این است که سیستم ذره‌ای که ما استفاده می‌کنیم فاقد اندازه است، بنابراین نمی‌توانیم با همین اطلاعات مشخص کنیم که چه زمانی جسم کاملا در آب فرو رفته است، اما می‌توانیم برای مولد نیروی شناوری آب یک اندازه مشخص در رابطه با ذره مشخص در نظر بگیریم و آن را به عنوان حداکثر عمق به کار ببندیم. بنابراین مولد برای فرو رفتن ذره بیش از آن مقدار نیروی بیشتری تولید نخواهد کرد. از طرف دیگر درست برعکس این موضوع نیز می‌باشد چرا که وقتی جسم به بالا حرکت می‌کند به جایی می‌رسد که فقط قسمتی از آن در آب است و قسمت دیگر بیرون از آب است، اما وقتی این حرمت را ادامه دهد به نقطه‌ای خواهد رسید که تمام آن از آب بیرون خواهد رفت، در این حالت اهمیت دارد که هیچ نیرویی از طرف این مولد نیرو ایجاد و به ذره وارد نشود و هرچقدر هم این بالا رفتن و فاصله گرفتن از آب بیشتر شود این موضوع باید صادق بماند. بنابراین می‌توان معادله محاسبه نیرو را به این شکل نشان داد:

$$f = \begin{cases} 0 & \text{when } d \leqslant 0 \\ v\rho & \text{when } d \geqslant 1 \\ dv\rho & \text{otherwise} \end{cases}$$

که در آن p چگالی مایع، v حجم شیئ و d مقدار عمقی است که شیئ در مایع فرورفته است که به صورت نسبتی از کل عمق فرورفتگی کامل جسم بیان می‌شود. وقتی کاملا جسم در مایع فرو رفته باشد، d بزرگتر یک و وقتی کاملا بیرون از مایع باشد d کوچکتر از صفر خواهد بود که از معادله زیر بدست می‌آید:

$$d = \frac{y_o - y_w - s}{2s}$$

که در آن s حدالثر عمق غرق شدگی جسم و y مختصات شیئ و مایع را مشخص می‌کند.

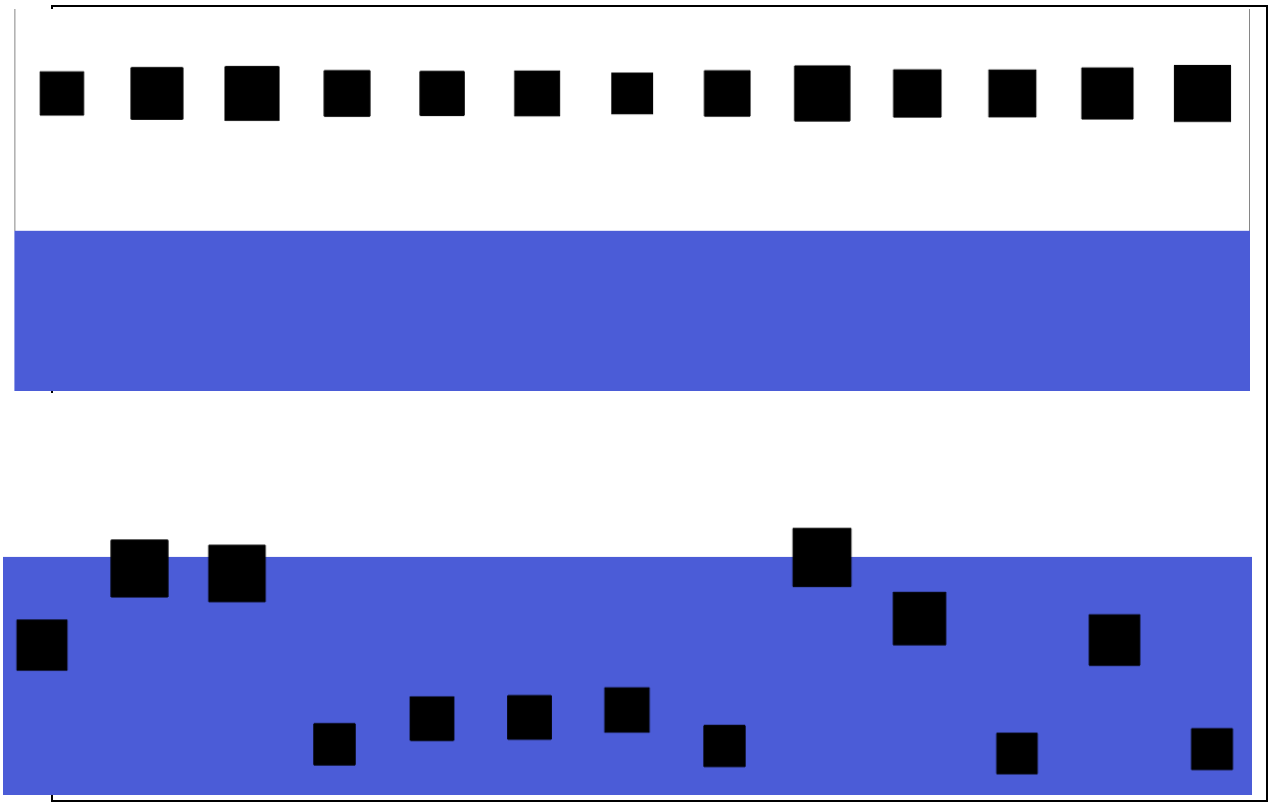
تمام این مباحث در کد زیر پیاده شده است:

```
function ParticleBuoyancy  (maxDepth,volume,waterHeight,liquidDensity){
   ParticleForceGenerator.call(this);
   this.maxDepth = maxDepth;
   this.volume = volume;
   this.waterHeight = waterHeight;
   this.liquidDensity = liquidDensity;
}
ParticleBuoyancy.prototype = new ParticleForceGenerator();
ParticleBuoyancy.prototype.constructor = ParticleBuoyancy;
ParticleBuoyancy.prototype.updateForce = function(particle,duration){
   var depth = particle.position.y;
   if (depth >= this.waterHeight + this.maxDepth){
      return;
   }
   var force = new Point(0,0);
   if (depth <= this.waterHeight - this.maxDepth)
   {
      force.y = this.liquidDensity * this.volume;
      particle.addForce(force);
      return;
```

```
    }
    force.y = this.liquidDensity * this.volume *(depth - this.maxDepth - this.waterHeight) / 2
* this.maxDepth;
    particle.addForce(force);
}
```

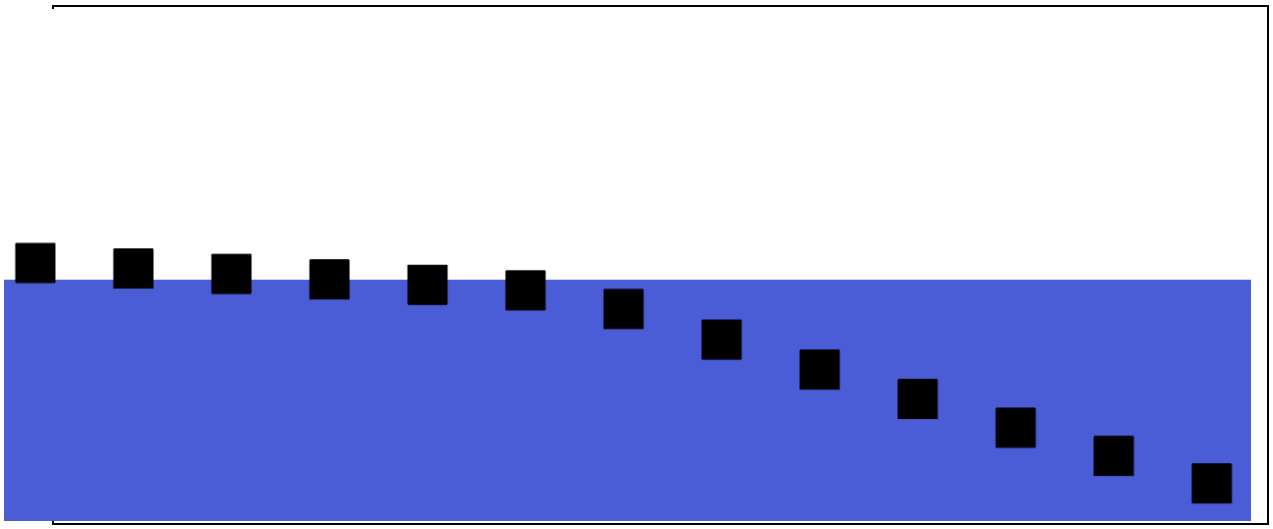
مولد بالا چهار پارامتر را دریافت می‌کند که عبارتند از بیشترین عمق فرورفتگی، حجم جسم، عمق از سطح آب و چگالی مایعی که جسم در آن غوطه ور است. با توجه به این پارامترها مشخص است که این مولد تنها به یک ذره می‌تواند اعمال شود و برای هر ذره باید مولد خاص خود را ایجاد نماییم.

در شکل‌های زیر یک مثال از مولد نیروی شناور آورده شده است که در آن تعدادی جسم با رفتار ذره از یک ارتفاع یکسان در یک مایع انداخته می شوند. تمام پارامترهای مربوط به نیروی مولد از جمله چگالی مایع، ارتفاع مایع و عیره یسان می باشند. آن چه برای هر جسم متفاوت است و به صورت رندم ایجاد می شود حجم هر جسم است که با توه با آن شکل آن نیز بزرگتر یا کوچکتر و در نتیجه ارتفاع فورفتگی آن هم متفاوت است، آن چنان که مشخص است، چگونه این پارامترها در رفتار مایع بر جسم شناور تاثیر با نسبت گفته شده می گذارد و به طور خلاصه نیروی:



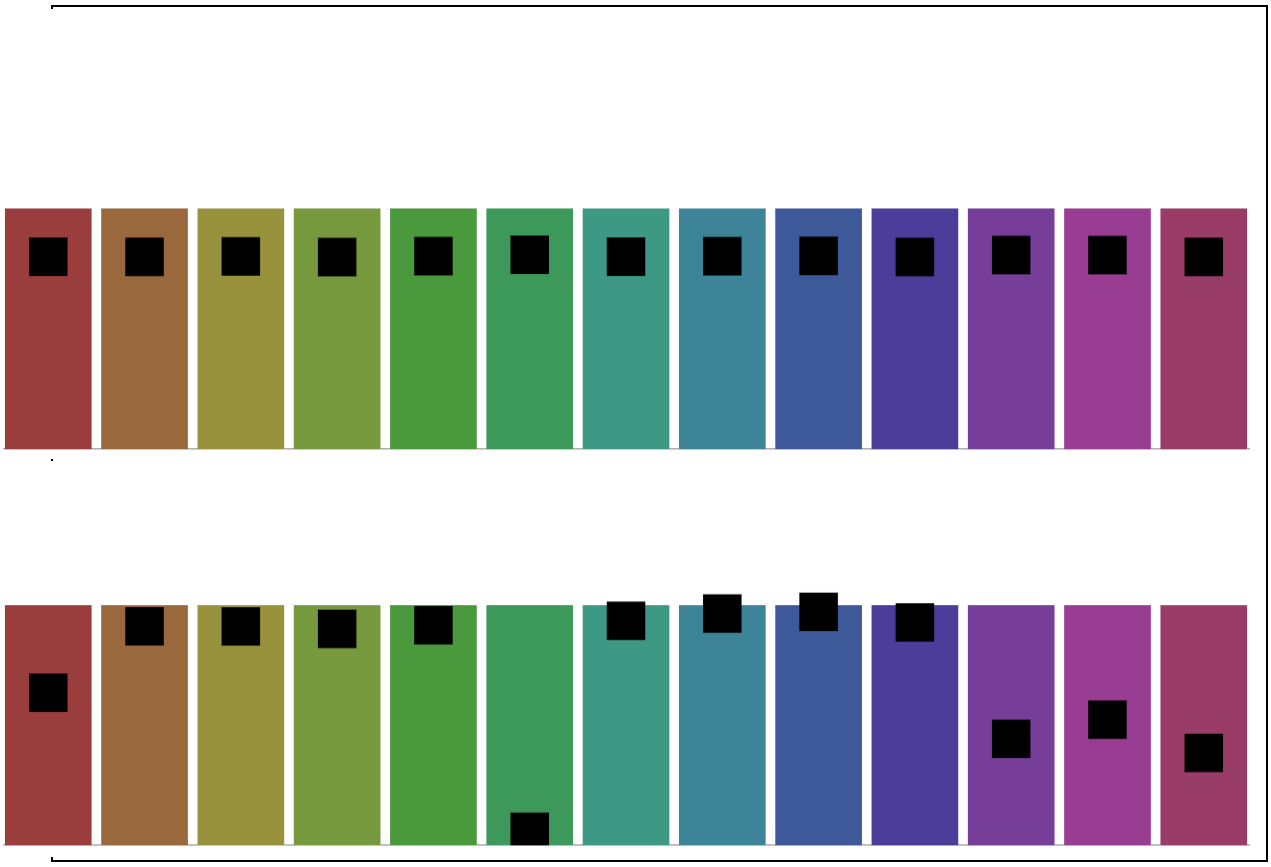
شکل ۹ مثال اول نیروهای شناوری

در یک مثال دیگر می توان مشاهده کرد که افزایش یک جسم ثابت با تمام متغیرهای دیگر ثابت چگونه در فرورفتگی جسم تاثیر گذار است. آن چنان که در تصویر زیر قابل مشاهده است، افزایش جرم جسم موجب بالا رفتن چگالی آن و در نتیجه بیشتر فرو رفتن جسم در آب می شود چنان که از یک حد به بعد این موضوع باعث فرو رفتن و غرق شدن کامل آن می شود.



شکل ۱۰ مثال سوم نیروهای شناوری

در مثال بعد تاثیر تغییر مایع و در نتیجه چگالی آن را بر جسم‌های کاملا یکسانی که از ارتفاع یکسانی در داخل مایع رها می شوند را مشاهده می کنیم که چگالی آن‌ها چطور بر رفتار یک جسم ثابت و نیروی وارد بر آن تاثیر می گذارد:



شکل ۱۱مثال دوم نیروهای شناوری

اما در یک مثال دیگر یک نوع استفاده دیگر از اجسام شناور و مایع کردیم و آن ایجاد موج‌هایی بر روی آب با استفاده از اجسام شناور به هم چسبیده است که این پدیده را به خوبی شبیه‌سازی می کنند، تصاویر این شبیه‌سازی در این جا آورده شده است:

شکل ۱۲مثال چهارم نیروهای شناوری

۴.۷ جمع‌بندی و نتیجه‌گیری

پس به طور خلاصه مشاهده کردیم که چگونه تعداد زیادی از پدیده‌های فیزیکی را می‌توان با خاصیت قانون هوک شبیه‌سازی کرد. بعضی از آن‌ها مانند نیروی شناوری کاملا مشابه نیروی فنری عمل می‌کنند. در نتیجه با توجه به این موضوع یک دسته از تامین کنندگان انرژی را پیاده‌سازی کردیم. لازم به ذکر است که می توان حتی برخوردها و نفوذ بسیاری دیگر از پدیده‌هایی که در فصل‌های بعد بررسی می‌کنیم را نیز با این قانون پیاده کرد اما از آنجا که قانون هوک و شبیه‌سازی آن در موارد جزئی ضعیف عمل می‌کند و در برخی موارد دقت لازم را ندارد این موارد را با ابزارهای دیگری پیاده می‌کنیم که در دو فصل بعد به آن‌ها می‌پردازیم.

فصل هشتم: محدودیت‌های قوی

۱.۸ مقدمه

با وجود این که با قانون فنر می‌توان وضعیت‌های مختلفی را شبیه‌سازی کرد باشد، در عین حال می تواند رفتار نامناسبی از خود نشان دهد. برای نمونه، شبیه‌سازی ضریب فنر، به هنگام بستن محکم اجسام به هم، تقریبا غیر ممکن است. در شرایطی که اجسام توسط میله‌های محکم به هم متصل می‌شوند و یا با استفاده از سطوح سخت از هم جدا نگه داشته شده اند، فنرگزینه‌ی بادوامی‌نیست.

در ابتدا نگاهی می‌اندازیم به معمول‌ترین محدودیت قوی، یعنی برخوردها و تماس بین اجسام. ریاضیات مربوط به این محدودیت، می‌تواند برای انواع دیگر محدودیت‌ها، همچون میله‌ها یا کابل‌های غیر قابل بسط که برای اتصال اجسام استفاده می‌شوند، به کار رود.

برای مدیریت محدودیت‌های قوی در موتور فیزیک پیاده‌سازی شده، نیاز به کنار گذاشتن دنیای مولدهای نیرو داریم. در تمام موتورهای فیزیک پیاده‌سازی شده در این پروژه، بین مولدهای نیرو و محدودیت های قوی تفاوت قائل شده است.

۲.۸ تحلیل ساده برخورد

برای مدیریت محدودیت‌های قوی، سیستمی‌به منظور تحلیل برخورد به موتور اضافه شده است. برخورد به هر وضعیتی که در آن دو جسم با هم درتماس باشند، گفته می‌شود. در زبان عامیانه برخورد به فرایندی خشن تلقی می‌شود که دو جسم در آن با سرعت نزدیک شونده قابل ملاحظه‌ای با هم مواجه می‌شوند. اما در نگاهی که در این موتور فیزیک به برخورد شده، علاوه بر این حالت، تماس دو جسم بدون سرعت نزدیک شونده نیز برخورد تلقی می‌شود. فرایندی مشابه برای تحلیل برخوردهای با سرعت زیاد و تماس ایستا به کار برده خواهد شد. به همین دلیل، در این جا دو لغت تماس و برخورد مفهوم یکسانی خواهند داشت.

زمانی که دو جسم با هم برخورد می‌کنند، حرکت بعد از برخورد این دو جسم می‌تواند با توجه به حرکت قبل از زمان برخورد محاسبه شود. این عمل، تحلیل برخورد نام دارد. برخورد به گونه‌ای تحلیل می‌شود که اطمینان حاصل شود که هر دو جسم بعد از برخورد، دارای حرکت صحیحی هستند. فرایند برخورد درمدت زمان بسیار کوتاهی رخ می‌دهد (به همین دلیل، ما قادر به دیدن فرایند برخورد بسیاری از اجسام نیستیم).

قوانین حاکم بر حرکت دو جسم در حال برخورد، بستگی به سرعت نزدیک شدن آن دو جسم دارد. سرعت نزدیک شدن، سرعت کلی است که دوجسم روی هم رفته، با آن سرعت حرکت می‌کنند. دقت شود که در این جا از لفظ سرعت نزدیک شدن (closing velocity) استفاده شد نه سرعت (speed)، چرا که سرعت کمیتی عددی است نه برداری، در نتیجه سرعت فاقد جهت است و تنها می‌تواند مقادیر مثبت یا صفر داشته باشد، این در حالی است که سرعت نزدیک شدن می‌تواند جهت داشته باشد. برای نمونه، دو جسمی‌که در جهت دور شدن از هم حرکت می‌کنند، سرعت نزدیک شدنی کمتر از صفر دارند. سرعت نزدیک شدن دو جسم با یافتن مولفه سرعت در جهت جسمی‌به جسم دیگر محاسبه می‌شود:

$$v_c = \dot{\mathbf{p}}_a \cdot \widehat{(\mathbf{p}_b - \mathbf{p}_a)} + \dot{\mathbf{p}}_b \cdot \widehat{(\mathbf{p}_a - \mathbf{p}_b)}$$

در رابطه‌ی بالا، v_c نشانگر سرعت نزدیک شدن (کمیتی مقداری) است، p_a و p_b موقعیت دو جسم a وb را نشان می‌دهند. نقطه(˙) نشان دهنده ضرب عددی، و p بردار طول واحد در جهت خود p است.

این معادله می‌تواند به صورت زیر ساده شود. عموما علامت این مقدار عوض می‌شود. در نتیجه معادله، به جای سرعت نزدیک شدن، سرعت دور شدن را مشخص می‌کند. سرعت نزدیک شدن برابر سرعت حرکت یک جسم نسبت به جسم دیگر، در جهت بین دو جسم است.

$$v_c = -(\dot{\boldsymbol{p}}_a - \dot{\boldsymbol{p}}_b) \cdot \widehat{(\boldsymbol{p}_a - \boldsymbol{p}_b)}$$

در معادله جدید، دو جسمی‌که در حال نزدیک شدن هستند دارای سرعت نسبی منفی و دو جسمی‌که در حال دور شدن هستند داری سرعت نسبی مثبت هستند. در واقع از نظر ریاضی، این کار معادل عوض کردن علامت در معادله قبل است.

زمانی که دو جسم با هم برخورد می‌کنند، به هم فشرده می‌شوند و تغییر شکل فنر مانند سطوح دوجسم، نیروهایی ایجاد می‌کند که باعث دور شدن دو جسم خواهد شد. تمام این اتفاقات در مدت زمان کوتاهی رخ می‌دهد (این اتفاقات آنقدر سریع رخ می‌دهد که امکان شبیه‌سازی فریم به فریم آن وجود ندارد). سرانجام دو جسم سرعت نزدیک شونده خود را از دست خواهند داد. با وجود این که این رفتار به مانند رفتار فنر است، اما در واقعیت اتفاقات دیگری نیز می‌افتد. در زمان فشرده شدن، هر اتفاقی می‌تواند رخ دهد وخصوصیات مواد درگیر، می‌تواند موجب بروز عکس العمل‌های پیچیده شود. در واقعیت این رفتار با رفتار فنر مطابقت ندارد.

مدل فنر فرض می‌کند که مقدار تکانه در طول مدت زمان برخورد، نگهداری می‌شود:

$$m_a\dot{\boldsymbol{p}}_a + m_b\dot{\boldsymbol{p}}_b = m_a\dot{\boldsymbol{p}}'_a + m_b\dot{\boldsymbol{p}}'_b$$

در این معادله، m_a نشان دهنده جرم جسم a، $\dot{p}_a$ سرعت جسم a قبل از برخورد، و $\dot{a'p}$ سرعت جسم a بعد از برخورد است.

خوشبختانه، بیشتر برخوردها رفتاری مشابه فنر ایده آل دارند. می‌توان با فرض نگه داشته شدن مقدار جنبش آنی، رفتاری کاملا قابل باور ایجاد کرد و از معادله قبل برای مدل کردن برخورد استفاده کرد.

معادله قبل، سرعت کل قبل و بعد از برخورد را مشخص می‌کند. اما از این رابطه نمی‌توان سرعت هر یک از اجسام را جداگانه محاسبه کرد. سرعت‌های جداگانه از طریق سرعت نزدیک شدن به هم مرتبط هستند. طبق معادله:

$$v_s' = -cv_s$$

که در آن، v'_s نمایانگر سرعت دور شدن بعد از برخورد، v_s سرعت دور شدن قبل از برخورد است و c ثابتی است به نام ضریب ارتجاع.

ضریب ارتجاع، سرعتی را که در آن، اجسام بعد از برخورد شروع به دور شدن می‌کنند، تحت کنترل دارد. این ضریب بستگی به مواد درگیر در برخورد دارد. دو ماده متفاوت، ضرایب متفاوتی دارند. برخی از اجسام مانند توپ تنیس، بعد از برخورد با راکت، به دور پرتاب می‌شوند. اما اجسام دیگر بعد از برخورد به هم می چسبند. مانندگلوله برفی پس از برخورد با صورت.

اگر این ضریب برابر یک باشد، اجسام با همان سرعتی که با آن به هم نزدیک شده بودند، به دور از هم پرتاب می‌شوند اما اگر این ضریب برابر صفر باشد، دو جسم یکی می‌شوند و با هم حرکت می‌کنند (یعنی سرعت دور شدشان برابر صفر است). حتی با صرف نظر از ضریب ارتجاع نیز، رابطه قبل‌تر همچنان برقرار است، یعنی مقدار کل تکانه همان خواهد بود. با استفاده از هر دو معادله، می‌توان مقادیر $\dot{p}_a$ و $\dot{p}_b$ را محاسبه کرد.

تا این جا در مورد مولفه‌های برخورد بین دو جسم صحبت شد. اما گاهی اوقات، قصد پیاده‌سازی برخورد بین یک جسم و چیزی که به طور فیزیکی قابل شبیه‌سازی نیست را داریم. این جسم ممکن است

زمین، دیوارهای یک سطح و یا هر جسم فاقد قدرت حرکت باشد. می‌توان این اجسام را به صورت اجسام با جرم بی نهایت نمایش داد ولی این کار سودی نخواهد داشت. مطابق تعریف این اجسام هرگز حرکت نمی‌کنند.

اگر بین یک جسم و قسمتی از یک صحنه فاقد حرکت برخوردی صورت گیرد، نمی‌توان سرعت دور شدن را بر اساس بردار بین مکان دو جسم محاسبه کرد، زیرا تنها یک جسم وجود دارد. به عبارت دیگر، نمی توان از لفظ (p_a-p_b)- در رابطه تکانه‌ها استفاده کرد. باید برای آن جایگزینی یافت. عبارت (p_a-p_b)- بیانگر جهت سرعت دور شدن است. سرعت دور شدن با استفاده از ضرب نقطه‌ای سرعت‌های نسبی دو جسم واین عبارت محاسبه می‌شود. اگر دو جسم وجود نداشته باشد، باید جهت به طور مشخص و صریح گفته شود. این جهت نشان دهنده جهتی است که دوجسم با هم برخورد می‌کنند و عموما به آن نرمال برخورد یا نرمال تماس گفته می‌شود. از آن جایی که این نرمال نشان‌دهنده‌ی جهت است، این بردار دارای بزرگی یک است.

در شرایطی که دو ذره داریم که با هم برخورد می‌کنند، نرمال تماس با استفاده از رابطه زیر به دست می‌آید.

$$\widehat{n} = \widehat{(p_a - p_b)}$$

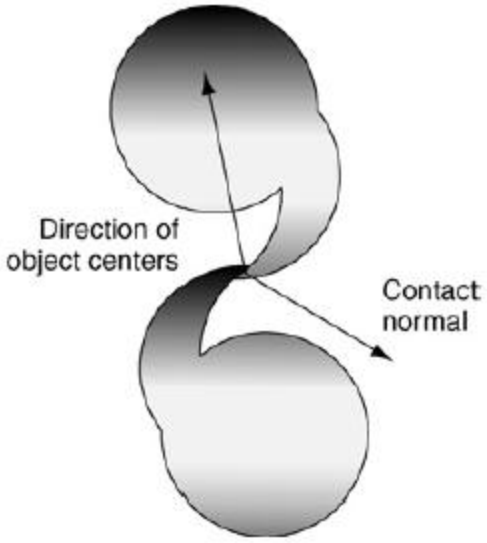
به طور قراردادی، همواره نرمال تماس را از زاویه دید جسم a در نظر گرفته می‌شود. در این مثال، از دید جسم a، تماس از جانب جسم b صورت می‌گیرد. در نتیجه از عبارت p_a-p_b استفاده می‌شود. اگرجهت برخورد از نگاه جسم b، در نظر گرفته شود، می‌توان به سادگی عبارت را در ۱- ضرب کرد.در عمل، از این روش به طور صریح استفاده نمی‌شود بلکه، عمل معکوس کردن را داخل کد مورد استفاده برای محاسبه سرعت دور شدن جسم b می‌آوریم. بعدا در بیان پیاده‌سازی این نکته قابل مشاهده است که علامتی منفی در محاسبات مربوط به جسم b آورده می‌شود.

زمانی که ذره‌ای با زمین بر خورد می‌کند، ما فقط جسم a را داریم و جسم b وجود ندارد. در این شرایط از دید جسم a، نرمال تماس

$$\hat{n} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

خواهد بود با این فرض که سطح در زمان برخورد هموار است.

اگر از ذرات بگذریم و بر روی اجسام صلب تمرکز کنیم، وجود نرمال تماس صریح، بسیار سخت خواهد بود. شکل زیر نمونه‌ای از شرایط ممکن را نشان می‌دهد. در این جا، دو جسم در حال برخورد، به موجب نوع شکلشان، نرمال تماسشان دقیقا در جهت عکس آن چیزی است که تصور می‌شود.. دو جسم به شکل قوس روی هم حرکت می‌کنند و نرمال تماس به گونه‌ای رفتار می‌کند که گویی مانع جدا شدن اجسام است تا تماس پیدا کردن. بعدا نمونه‌های دیگری آورده می‌شود که می‌تواند برای نمایش میله و یا اتصالات سخت دیگر به کار رود.



شکل۱۳ خطای نرمال

با داشتن نرمال تماس صحیح، معادله سرعت نزدیکی به صورت زیر در می‌آید:

$$v_s = (\dot{p}_a - \dot{p}_b) \cdot \hat{n}$$

تنها تغییری که برای تحلیل برخورد لازم است، تغییر سرعت است. تا این جا در این موتور فیزیک، تنها روشی که برای تغییر سرعت به کار برده شده است، تغییر شتاب بوده است. اگر تغییر شتاب برای مدت زمانی طولانی اعمال شود، سرعت به شدت تغییر خواهد کرد. اما در مورد مطرح شده در این جا، تغییرات لحظه‌ای هستند. به همین دلیل سرعت نیز به صورت آنی مقادیرجدید به خود می‌گیرد. لازم به ذکر است که اعمال نیرو سبب تغییر شتاب یک جسم می‌شود. اگر ما نیرو را فورا تغییر دهیم شتاب نیز فورا تغییر خواهد کرد. در رابطه با تغییر سرعت نیز می‌توان چیزی مشابه در نظر گرفت. اما به جای نیرو به آن ضربه گفته می‌شود. ضربه یعنی تغییری آنی و فوری در سرعت. همان گونه که در مورد نیرو رابطه‌ی برقرار است:

$$f = m\ddot{p}$$

در رابطه با ضربه نیز رابطه زیرصدق می‌کند:

$$g = m\dot{p}$$

یک تفاوت اساسی بین نیرو و ضربه وجود دارد. جسم فاقد شتاب است مگر این که نیرویی به آن وارد شود یعنی می‌توان شتاب کل را با استفاده از قانون دالامبر از طریق ترکیب تمام نیروها محاسبه کرد. اما اگر به جسم نیرو یا ضربه‌ای نیز وارد نشود باز هم سرعت خواهد داشت. ضربه سرعت را تغییر می‌دهد. می‌توان ضربه را نیز با استفاده از قانون دالامبر ترکیب کرد. اما نتیجه برابر مجموع تغییرسرعت خواهد بود نه کل سرعت:

$$\dot{p}' = \dot{p} + \frac{1}{m}\sum_{n} g_i$$

g_1 تا g_n مجموعه‌ی تمام ضربه‌های اعمال شده بر جسم است. در واقعیت، ضربه‌ها را به گونه‌ای که نیروها را جمع کردیم، جمع نمی‌کنیم. ضربه‌ها در طول فرایند برخورد، هر زمان که رخ دهند، اعمال می‌شوند. هر یک از ضربه‌ها در زمانی خاص با استفاده از معادله زیر، اعمال می‌شوند.

$$\dot{p}' = \dot{p} + \frac{1}{m}g$$

نتیجه‌ی تحلیل برخورد، ضربه‌ای است که بر هر جسم وارد می‌شود. ضربه فورا بر جسم اعمال می‌شود و سرعت به طور آنی تغییر می‌کند.

به منظور مدیریت برخوردها از کد زیر استفاده می‌کنیم. وظیفه lvertactResoCon گرفتن مجموعه‌ای ازبرخوردها و اعمال ضربه‌های مناسب به اجسام درگیر در برخورد است. هر برخورد در یک ساختمان داده Contact آماده شده است که به شکل زیر است:

```
function ParticleContact()
{
   this.particle = new Array();
   this.restitution = null;
   this.contactNormal = new Point();
}
```

این ساختمان داده اشاره گری به هر یک از اجسام درگیر برخورد نگه می‌دارد: یک بردار که حاوی نرمال تماس از دید جسم اول است و عضو داده‌ای برای نگه داری ضریب ارتجاع تماس. اگر برخورد بین یک جسم و یک صحنه رخ دهد (تنها یک جسم وجود داشته باشد)، در این صورت اشاره گر جسم دوم برابر NULL خواهد بود.

برای تحلیل یک تماس، معادلات برخوردی که قبلا ملاحظه شد را پیاده‌سازی می‌کنیم.

```
ParticleContact.prototype.resolve = function(duration){
   this.resolveVelocity(duration);
}
ParticleContact.prototype.calculateSeparatingVelocity = function(){
   var relativeVelocity = this.particle[0].velocity;
   if (this.particle[1]){
      relativeVelocity.x -= this.particle[1].velocity.x;
      relativeVelocity.y -= this.particle[1].velocity.y;
   }
   return relativeVelocity.scalarProduct(this.contactNormal);
}
ParticleContact.prototype.resolveVelocity = function(duration){
   var separatingVelocity = this.calculateSeparatingVelocity();
   if (separatingVelocity > 0)
   {
      return;
   }
   var newSepVelocity = -separatingVelocity * this.restitution;
```

```
    var deltaVelocity = newSepVelocity - separatingVelocity;
    var totalInverseMass = this.particle[0].inverseMass;
    if (this.particle[1]){
        totalInverseMass += this.particle[1].inverseMass;
    }
    if (totalInverseMass <= 0) return;
    var impulse = deltaVelocity / totalInverseMass;
    var impulsePerIMass = new Point(this.contactNormal.x * impulse,this.contactNormal.y *
impulse);
    this.particle[0].velocity.x = this.particle[0].velocity.x + impulsePerIMass.x *
this.particle[0].inverseMass;
    this.particle[0].velocity.y = this.particle[0].velocity.y + impulsePerIMass.y *
this.particle[0].inverseMass;
    if (this.particle[1])
    {
        this.particle[1].velocity.x = this.particle[1].velocity.x + impulsePerIMass.x * -
this.particle[1].inverseMass;
        this.particle[1].velocity.y = this.particle[1].velocity.y + impulsePerIMass.y * -
this.particle[1].inverseMass;

    }
}
```

این کد، سرعت هر یک از اجسام درگیر را به منظور انعکاس برخورد، تغییر می‌دهد.

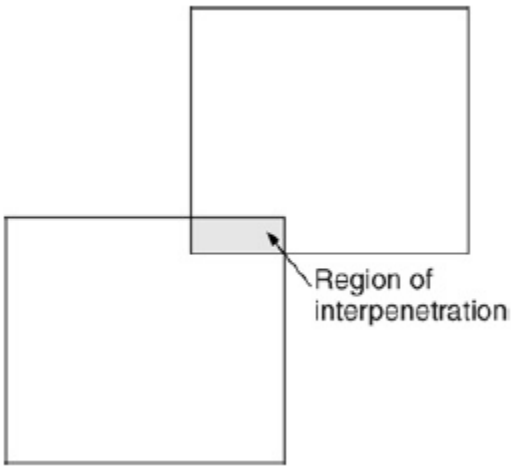
نقاط برخورد معمولا توسط یک یابنده برخورد مشخص می‌شود. یابنده برخورد تکه کدی است که مسئول یافتن دو جسم در حال برخورد یا جسم در حال برخورد با یک صحنه‌ی فاقد قابلیت حرکت است. در این موتور فیزیک، نتیجه‌ی الگوریتم تشخیص برخورد، مجموعه‌ای از ساختمان داده‌های Contact است که با اطلاعات مناسب پر شده است. تشخیص برخورد نیاز به دانستن هندسه اجسام، یعنی شکل و اندازه اجسام دارد. تا این جا در این موتور فیزیک، اجسام به صورت ذره فرض شده است که اصلا هندسه‌ای برای آن در نظر گرفته نشده است.

سیستم شبیه‌ساز فیزیکی (قسمتی از موتورکه مسئولیت مدیریت قوانین حرکت، تحلیل برخورد و نیروها را بر عهده دارد)، نیازی به آگاهی از جزئیات شکل اجسام درگیر برخورد ندارد. سیستم تشخیص برخورد

مسئولیت محاسبه هرگونه خصوصیت هندسی را بر عهده دارد. برای نمونه تشخیص این که دو جسم درچه زمان و مکانی با هم تماس پیدا می‌کنند و مشخص کردن نرمال تماس بین دو جسم.

برخی از الگوریتم‌های تشخیص برخورد، شیوه‌ی حرکت اجسام را زیر نظر می‌گیرند و تلاش می‌کنند برخوردهای احتمالی آینده را پیش بینی کنند. به این صورت که به مجموعه اجسام نگاه می‌اندازند و بررسی می کنند که آیا هیچ دو جسمی‌از هم خواهند گذشت یا خیر.

دو جسم با هم تداخل دارند اگر به صورت جزئی به مانند شکل زیر در هم ادغام شده باشند:



شکل ۱۴ تداخل

در زمان پردازش برخوردی که دو جسم درهم فرو می‌روند، تنها تغییر سرعت کافی نیست. اگر اجسام در این حالت با ضریب ارتجاع نسبتا کوچکی با هم برخورد کنند، سرعت دورشدنشان تقریبا برابر صفر خواهد بود.در این حالت دو جسم هیچ گاه از هم جدا نمی‌شوند و فرد بازیکن مشاهده می‌کند که اجسام در هم گیر کرده اند و جداسازی آن‌ها غیر ممکن خواهد بود.

به عنوان جزیی از تحلیل برخورد، نیاز به تحلیل تداخل داریم. وقتی دو جسم در هم نفوذ می‌کنند، تنها به اندازه‌ای دو جسم را از هم دور می‌کنیم که از هم جدا شوند. این وظیفه یابنده برخورد است که به عنوان بخشی از ساختمان داده Contact مشخص کند دو جسم چقدر در هم نفوذ کرده‌اند. محاسبه مقدار نفوذ دو جسم بستگی به هندسه اجسام درگیر در برخورد دارد. همان طور که دیده شد این کار وظیفه سیستم یابنده برخورد است نه شبیه‌ساز فیزیک.

برای نگه داشتن این اطلاعات، یک عضو داده‌ای به ساختمان داده Contact به صورت زیر اضافه می شود:

```
this.penetration = null;
```

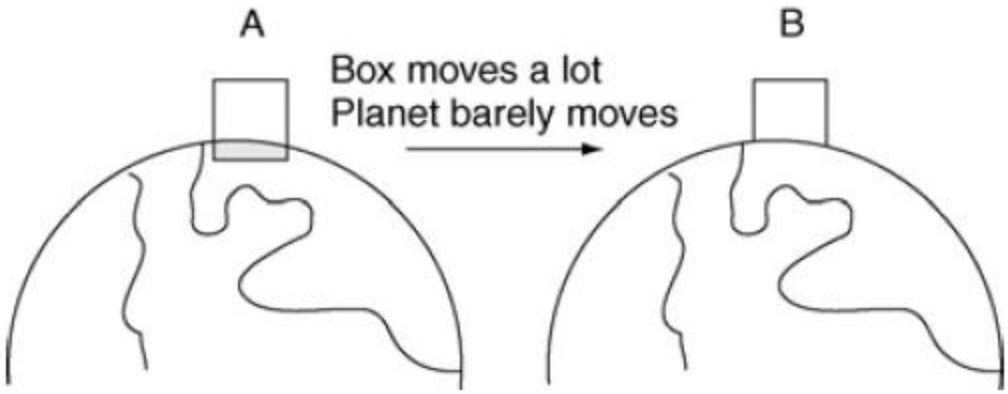
توجه داشته باشید که همانند سرعت نزدیک شدن، عمق نفوذ هم دارای بزرگی است و هم علامت. عمق نفوذ منفی بیانگر آن است که دو جسم هرگز از هم نمی‌گذرند. عمق نفوذ صفر نشان‌دهنده‌ی این است که دو جسم فقط هم دیگر را لمس می‌کنند.

به منظور تحلیل تداخل، تنها نیاز به کنترل عمق نفوذ داریم. در صورتی که مقدار عمق نفوذ صفر یا کمتر از صفر باشد، نیاز به انجام هیچ اقدامی‌نیست. در غیر این صورت اجسام را فقط به اندازه‌ای می‌توانیم از هم دور کنیم که عمق نفوذ برابر صفر شود. عمق نفوذ باید درجهت نرمال تماس داده شود. اگر اجسام را درجهت نرمال تماس به اندازه عمق نفوذ جابه جا کنیم، آنگاه دو جسم دیگر با هم در تماس نخواهند بود. زمانی که فقط یک جسم در تماس دخالت دارد (تداخل با صحنه بازی رخ می‌دهد) نیز، همین روش انجام می‌شود یعنی عمق نفوذ در جهت نرمال تماس خواهد بود.

حال که مقدار جابه‌جایی کل دو جسم (مقدار نفوذ) و جهتی که اجسام باید جابه‌جا شوند مشخص شده است، مقدار جابه‌جایی جداگانه هر یک از اجسام باید مشخص شود.

اگر تنها یک جسم در تماس درگیر باشد، راه حل ساده است: جسم باید به اندازه کل فاصله تداخل، جابه‌جا شود. اگر دو جسم وجود داشته باشد، دراین صورت طیف گسترده‌ای از راه حل‌ها وجود دارد. برای مثال، می‌توان به سادگی هر یک از دو جسم را به اندازه نصف مقدار نفوذ جابه جا کرد. این راه در برخی از وضعیت‌ها قابل انجام است اما مشکل باورپذیری ایجاد می‌کند. برای نمونه، جعبه‌ای کوچک را که بر روی سطح سیاره‌ای را شبیه‌سازی شده است در نظر بگیرید، اگر جعبه مقداری درداخل سیاره نفوذ کرده باشد، آیا جعبه و سیاره باید به اندازه مساوی جابه‌جا شوند ؟

باید به این نکته توجه کرد که درابتدا نفوذ چگونه به وجود آمده است و در واقعیت چه اتفاقی رخ می دهد. شکل ۱۵، حالت A، جعبه و سیاره را در حالت نفوذ نشان می‌دهد. باید تا جایی که ممکن است به حالت B تصویر نزدیک شویم، به گونه‌ای که گویی فیزیک واقعی در جریان است.



شکل۱۵ مشکل تداخل در شکل‌های بزرگ

برای انجام این عمل، اجسام را به نسبت عکس جرمشان جابه جا می‌کنیم. جسمی‌با جرم زیاد بسیار کم جابه جا می‌شود و جسمی‌با جرم ناچیز باید به مقدار زیادی جایه جا شود. اگر یکی از دو جسم دارای جرم بی نهایت باشد، اصلا جابه جا نمی‌شود و جرم دیگر باید تمام جابه‌جایی را انجام دهد.

جمع کل مقدار جابه‌جایی برابر مقدار تداخل است:

$$\Delta p_a + \Delta p_b = d$$

Δp_a برابر فاصله‌ای است که جسم a باید طی کند. این دو فاصله از طریق نسبتی که با جرم اجسام دارند به هم مرتبط هستند.

$$m_a \Delta p_a = m_b \Delta p_b$$

این دو رابطه اگر با هم ترکیب شوند معادلات زیر به دست می‌آید:

$$\Delta p_a = \frac{m_b}{m_a + m_b} d$$

$$\Delta p_b = \frac{m_a}{m_a + m_b} d$$

با ترکیب این دو با جهت نرمال تماس، تغییر کل در جهت برداری به دست می‌آید:

$$\Delta \boldsymbol{p}_a = \frac{m_b}{m_a + m_b} d\boldsymbol{n}$$

$$\Delta \boldsymbol{p}_b = -\frac{m_a}{m_a + m_b} d\boldsymbol{n}$$

n نشان دهنده نرمال تماس است (علامت منفی موجود در رابطه دوم به دلیل این است که نرمال تماس از دید جسم a محاسبه شده است.).

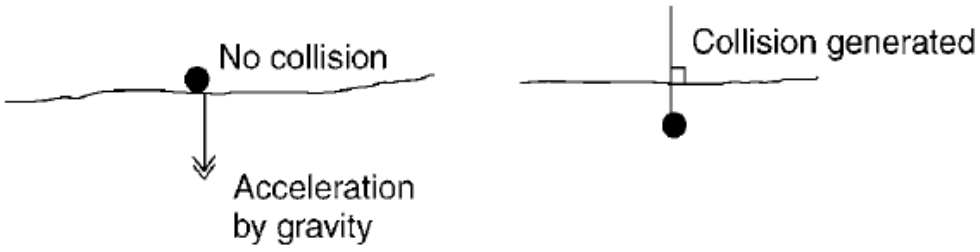
پیاده‌سازی معادله تحلیل تداخل به صورت زیر است:

```
ParticleContact.prototype.resolveInterpenetration = function(duration){
   if (this.penetration <= 0) return;
   var totalInverseMass = this.particle[0].inverseMass;
   if (this.particle[1]){
      totalInverseMass += this.particle[1].inverseMass;
   }
   if (totalInverseMass <= 0) return;
   var movePerIMass = new Point(this.contactNormal.x *(-this.penetration /
totalInverseMass),this.contactNormal.y *(-this.penetration / totalInverseMass));
   this.particle[0].x = this.particle[0].x + movePerIMass.x * this.particle[0].inverseMass;
   this.particle[0].y = this.particle[0].y + movePerIMass.y * this.particle[0].inverseMass;
   if (this.particle[1])
   {
      this.particle[1].x = this.particle[1].x + movePerIMass.x * this.particle[1].inverseMass;
      this.particle[1].y = this.particle[1].y + movePerIMass.y * this.particle[1].inverseMass;
   }
}
```

تا این جا پیاده‌سازی انجام شده، تغییر سرعت انجام شده در زمان برخورد و تحلیل تداخل را اعمال می کند. حال با اجرای سیستم تحلیل تماس ملاحظه می‌شود که این سیستم، برای برخوردهای با سرعت متوسط به خوبی عمل می‌کند. اما اجسام ایستا و ساکن (برای مثال ذره‌ای که برروی یک میز به صورت ساکن قرار دارد)، ممکن است شروع به لرزش کنند وحتی برخی اوقات به هوا برمی‌خیزند. برای داشتن یک سیستم تحلیل تماس کامل و پایدار، نیاز به بازبینی اتفاقاتی که در زمان برخورد دو جسم با سرعت نزدیک شدن کم یا صفر رخ می‌دهد، داریم.

در این موتور فیزیک، سیستم تحلیل تماس برای برخوردهای با سرعت متوسط طراحی شده است. شبیه‌ساز فیزیکی که در این جا ارائه شد، توانایی کنترل این شرایط را دارد. اما یابنده‌های برخورد ممکن است در این شرایط نتایج عجیبی تولید کنند. برای مثال، ممکن است قبل از این که یابنده برخورد تشخیص دهد که دو جسم با هم تماس داشته اند، دو جسم از هم بگذرند. این امکان وجود دارد که در زمانی که یابنده برخورد بتواند برخورد را تشخیص دهد، دو جسم تا نیمه از هم گذشته باشند و در حال جدا شدن از هم باشند، در این شرایط اجسام دارای سرعت دور شدن مثبت هستند و هیچ ضربه‌ای ایجاد نمی‌شود.

شرایط موجود در تصویر زیر را در نظر بگیرید.



شکل ۱۶ مشکل برخوردهای ایستا

ذره‌ای ساکن بر روی زمین قرار دارد. تنها نیرویی که به آن وارد می‌شود نیروی جاذبه زمین است. در اولین فریم، ذره شتابی به سوی پایین می‌گیرد. سرعت ذره افزایش می‌یابد ولی موقعیت ذره ثابت می‌ماند (در ابتدای فریم، ذره فاقد سرعت است.). در فریم دوم، موقعیت به روز می‌شود و سرعت دوباره افزایش می‌یابد. حال ذره در حال حرکت به سمت پایین است و در حال تداخل با زمین است. یابنده برخورد نفوذ را تشخیص می دهد، تحلیلگر تماس ذره را مورد بررسی قرار می‌دهد و متوجه می‌شود که ذره دارای سرعت نفوذ زیر است:

$$\dot{p} = 2\ddot{p}t$$

با اعمال پاسخ برخورد، ذره سرعتی برابر

$$\dot{p}' = c\dot{p} = c2\ddot{p}t$$

پیدا می‌کند و از حالت تداخل خارج می‌شود. در فریم سوم، ذره دارای سرعتی رو به بالا است که موجب می‌شود جسم از زمین جدا شود و به هوا برود. سرعت رو به بالا بسیار کم است، اما به اندازه‌ای که است که می توان متوجه حرکت آن شد. در واقع، اگر فریم اول یا دوم به طور غیر معمولی طولانی باشد، سرعت رو به بالا این امکان را دارد که به طرز قابل توجه‌ای افزایش یابد و ذره را به سرعت به آسمان بفرستد. اگر این الگوریتم برای بازی با نرخ فریم متغیر پیاده‌سازی شود و به تدریج نرخ فریم را کاهش دهیم (برای نمونه، با انجام کاری دیگر در زمینه تصویر)، هر جسم ساکنی به طور ناگهانی می‌پرد.

برای حل این مشکل، دو روش وجود دارد. در روش اول، نیاز به تشخیص زود هنگام تماس داریم. در این مثال، پس از گذشت دو فریم، متوجه وجود مشکل شدیم. اگر یابنده برخورد را به گونه‌ای تنظیم کنیم که تماس‌هایی که نزدیک هستند اما تداخل کمی‌دارند را باز گرداند، می‌توان بعد از گذشت فریم اول، تماس را مدیریت کرد.

در روش دوم، باید زمانی را که جسم دارای سرعتی است که تنها ناشی از نیروهای وارد شده بر آن در یک فریم است، تشخیص دهیم. پس از اتمام فریم اول، سرعت جسم تنها ناشی از نیروی جاذبه وارد بر آن در یک فریم است. می‌توان سرعت در انتهای فریم را به سادگی از طریق ضرب نیرو در طول مدت زمان فریم به دست آورد. اگر سرعت واقعی ذره، مساوی و یا کمتر از این مقدار باشد (یا حتی مقدار ناچیزی بیشتر از این سرعت)، می‌توان متوجه شد که ذره در فریم قبلی ساکن بوده است. در این حالت، تماس احتمالا تماسی ایستا خواهد بود نه تماس برخوردی و به جای به کار بردن محاسبات ضربه مربوط به برخورد، ضربه‌ای به ذره اعمال می‌کنیم که موجب شود ذره سرعت دور شدنی برابر صفر به خود بگیرد.

به طور کلی در یک تماس ایستا، مسائل زیر رخ می‌دهد: در ابتدا فرصتی برای افزایش سرعت دور شدن وجود نخواهد داشت، در نتیجه بعد از تماس، سرعت دور شدن برابر صفر خواهد بود. در مثال بالا، سرعت به روش به کار برده شده در تقسیم زمان به تعدادی فریم، وابسته است، در نتیجه می‌توان با جسم به گونه‌ای رفتار کرد که گویی قبل از تماس دارای سرعت صفر بوده است. ذره‌ای که دارای سرعت صفر است و در مثال بالا ذره همواره در فریم اول باقی می‌ماند.

از نقطه نظر دیگری نیز می‌توان به این مسئله نگاه گرد. در واقع، در هر فریم، شاهد برخوردی با ضریب ارتجاع صفر هستیم. این مجموعه از برخوردهای جزیی، اجسام را ازهم جدا نگه می‌دارد. از این جهت، به موتوری که قابلیت مدیریت تماس‌های ایستا را دارد، "موتور برخورد جزیی" نیز گفته می‌شود.

هنگامی‌که دو جسم در وضعیت تماس ایستا قرار دارند، سرعت نسبی دو جسم از سرعت مطلق هر یک از دو جسم، از اهمیت بالاتری برخوردار است. دو جسم ممکن است با هم در یک جهت در تماس ایستا باشند، اما نسبت به هم، درجهت دیگر حرکت کنند. یک جعبه ممکن است بر روی زمین ساکن قرار داشته باشد، در حالی که ممکن است هم زمان بر روی سطح در حال لغزش باشد. در این جا به کدی به منظور مدیریت تماس های لرزشی نیاز است که بتواند اجسامی‌را که بر روی هم سر می‌خورند، کنترل نماید. این بدین معنی است که دیگر امکان استفاده از سرعت مطلق هر جسم، به طور جداگانه وجود ندارد.

برای مدیریت این وضعیت، محاسبات مربوط به سرعت و شتاب تنها در جهت نرمال تماس صورت می گیرد. ابتدا سرعت را در این جهت به دست می‌آوریم، سپس بررسی می‌کنیم که آیا این سرعت تنها توسط مولفه شتاب درجهتی یکسان ایجاد شده است یا خیر. اگر این گونه باشد، سرعت تغییر می‌کند، بنابراین سرعت دور شونده یا نزدیک شونده در این جهت وجود ندارد. اما هنوز این امکان وجود دارد که سرعتی نسبی در جهات دیگر موجود باشد، با این وجود از آن صرف نظر می‌شود.

پیاده‌سازی این مورد خاص، به صورت زیر به تابع پردازش برخورد اضافه می‌شود:

```
   var accCausedVelocity = this.particle[0].acceleration;
   if (this.particle[1]){
      accCausedVelocity.x -= this.particle[1].acceleration.x;
      accCausedVelocity.y -= this.particle[1].acceleration.y;
   }
   var accCausedSepVelocity = accCausedVelocity.scalarProduct(this.contactNormal) *
duration;
   if (accCausedSepVelocity < 0)
   {
      newSepVelocity += this.restitution * accCausedSepVelocity;
      if (newSepVelocity < 0) newSepVelocity = 0;
   }
```

به منظور نگه داشتن دو جسم در وضعیت تماس ایستا، در هر فریم تغییر کمی‌در سرعت اعمال می شود. این تغییر به منظور تصحیح افزایش سرعتی که از فرو رفتن دو جسم در هم در طول یک فریم به وجود می‌آید، صورت می‌گیرد.

روش برخورد جزیی که در بالا به آن اشاره شد، تنها یکی از روش‌های ممکن در مدیریت تماس ایستا است. تماس ایستا یکی از دو چالش موجود در طراحی یک موتور فیزیک است (چالش دیگر اصطکاک است که در واقع این دو چالش، با هم مطرح می‌شوند.). روش‌های حمله، پیچیدگی‌ها و بی ثباتی‌های بسیاری وجود دارد.

یک روش فیزیکی واقع گرایانه، توجه به این نکته است که در واقعیت نیرویی از جانب زمین بر ذره اعمال می‌شود. این نیروی واکنش، جسم را به عقب فشار می‌دهد تا شتاب کل جسم در جهت عمودی برابر صفر شود. جسم هر چقدر به سمت پایین نیرو وارد کند، زمین به همان میزان در جهت بالا نیرو وارد می‌کند. می‌توان چنین مولد نیرویی ایجاد کرد تا اطمینان حاصل شود که هیچ شتابی به سمت زمین وجود ندارد.

این روش در مورد ذره‌هایی که فقط یک تماس با زمین دارند به خوبی عمل می‌کند. اما وضعیت درارطبه با اجسام صلب و سخت پیچیده‌تر خواهد بود. این امکان وجود دارد که چندین نقطه تماس بین جسم و

نقاط ساکن غیر قابل حرکت وجود داشته باشد. این امکان وجود ندارد که به صورت آنی نیروهای واکنش را در هر نقطه تماس محاسبه کرد به گونه‌ای که حرکت کلی جسم صحیح باشد.

تحلیلگر تماس، لیستی ازتماس‌ها را از سیستم تشخیص برخورد، دریافت می‌کند و اجسام در حال شبیه‌سازی را به روز می‌کند تا بتواند تماس‌ها را کنترل نماید. برای انجام این به روز رسانی، سه دسته کد وجود دارد:

۱ـ تابع تحلیل برخورد که به اجسام، ضربه اعمال می‌کند تا دور شدن آن‌ها را شبیه‌سازی کند.

۲ـ تابع تحلیل تداخل که اجسام را از هم دور می‌کند تا جایی که دیگر با هم تداخل نداشته باشند.

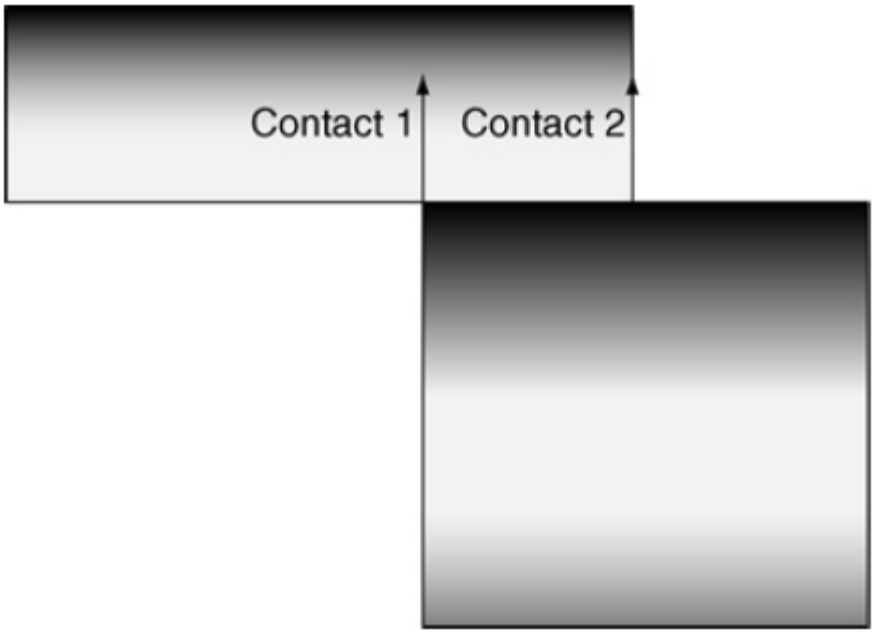
۳ـ کد مربوط به تماس ایستا که در داخل تابع تحلیل برخورد قرار دارد و تماس‌هایی را مدیریت می‌کند که بیشتر احتمال دارد ایستا باشند و نه برخورد.

این مسئله که کدام تابع نیاز به فراخوانی تماس دارد، بستگی به سرعت دور شدن و مقدار تداخل دارد. تحلیل تداخل تنها زمانی انجام می‌شود که تماس دارای عمق نفوذی با مقدار بزرگتر از صفر باشد. همچنین، در صورتی که دو جسم در هم نفوذ کرده باشند اما جدا از هم باشند، ممکن است تنها نیاز به اجرای تحلیل تداخل باشد و نیازی به تحلیل برخورد نباشد.

با صرف نظر از ترکیب توابع مورد نیاز، هر تماسی به طور جداگانه تحلیل می‌شود. این عمل با هدف ساده کردن دنیای واقعی انجام می‌شود. برخی از تماس‌ها، به صورت سلسله مراتبی اثر می‌کنند، اما برخی دیگر، ترکیب می‌شوند و همزمان روی جسم اثر می‌کنند. برخی همین روند را تکرار می‌کنند، تماس‌های متوالی را به ترتیب و تماس‌های ایستا را به صورت همزمان تحلیل می‌کنند.

در پیاده‌سازی این موتور جستجو، نگاهی ساده به مسائل شده است و در پایان هر فریم تمام تماس‌ها به طور همزمان تحلیل نشده است. با این وجود، برای رسیدن به نتایج دقیق تر، نیاز به تحلیل تماس‌ها به ترتیب صحیح است.

اگر جسمی، همانگونه که در تصویر زیر نیز نشان داده شده است، دارای دو تماس همزمان باشد، تغییر سرعت آن جسم به منظور تحلیل یکی از تماس‌ها، ممکن است سرعت دور شدن در تماس دیگر را تغییر دهد. در این تصویر، اگر اولین تماس تحلیل شود، دومین تماس دیگر برخورد تلقی نخواهد شد، یعنی جدا می‌شود. اما اگرابتدا تماس دوم تحلیل شود، تماس اول همچنان نیاز به تحلیل شدن دارد و تغییر سرعت کافی نخواهد بود.



شکل۱۷ مشکل اولویت تداخل‌ها

به منظور جلوگیری از این چنین اعمال غیر ضروری، ابتدا شدیدترین و سخت‌ترین تماس را باید تحلیل کرد: یعنی تماسی که کمترین سرعت دورشدن (منفی‌ترین) را دارد. علاوه بر راحت بودن، این عمل بهترین عمل فیزیکی واقع گرایانه قابل انجام خواهد بود. در تصویر، اگر رفتار وضعیتی که هر سه جسم درآن قرار دارند را با

رفتار وضعیتی که یکی از دو بلوک زیری برداشته شده است مقایسه کنیم، نتیجه نهایی بسیار مشابه حالتی است که بلوک A وجود دارد اما بلوک B وجود نداشته باشد. به عبارت دیگر، سخت‌ترین برخوردها، رفتار شبیه‌سازی را تحت سلطه خود دارند. اگر برای مدیریت برخوردها نیاز به اولویت بندی باشد، اولویت با واقعی ترین برخورد خواهد بود. تصویر بالا نشان دهنده پیچیدگی الگوریتم تحلیل تماس است. اگر یکی از برخوردها مدیریت شود، سرعت دور شدن تماس‌های دیگر امکان دارد تغییر یابد. نمی‌توان تنها با مرتب کردن تماس‌ها بر اساس سرعت دور شدنشان، به ترتیب آن‌ها را تحلیل کرد. چرا که با تحلیل اولین برخورد، ممکن است سرعت دور شدن تماس بعدی مثبت شود و دیگر نیازی به پردازش نباشد.

مشکل ظریف دیگری نیز وجود دارد که در بسیاری از وضعیت‌ها رخ نخواهد داد. برای مثال، ممکن است با وضعیتی رو به رو شویم که تحلیل تماس دوم، تماس اول را دوباره به برخورد بازگرداند، در نتیجه نیاز به تحلیل دوباره تماس اول داریم. خوشبختانه، می‌توان نشان داد که در برخی از انواع شبیه‌سازی (به خصوص شبیه‌سازی‌های فاقد اصطکاک)، این حلقه تدریجا به جواب صحیح ختم می‌شود و نیازی به ادامه حلقه تا ابد نیست و در پایان با وضعیتی که اصلاحات تا اندازه‌ای بزرگ شود که شبیه‌سازی منفجر شود، مواجه نخواهیم شد. اما این روش ممکن است مدت زیادی زمان ببرد و هیچ راهی برای تخمین زمان لازم وجود ندارد. به همین دلیل، محدودیتی بر روی تعداد تحلیل‌هایی که در هر فریم می‌تواند انجام شود، اعمال می‌کنیم.

تحلیلگر تماس مورد استفاده در این پروژه، از الگوریتم زیر استفاده می‌کند:

۱ـ محاسبه سرعت دور شدن هر تماس و مشخص کردن تماسی با کمترین (منفی‌ترین) مقدار.

۲ـ اگر کم‌ترین سرعت دور شدن، بزرگ‌تر و یا مساوی صفر باشد، از الگوریتم خارج می‌شویم.

۳ـ پردازش الگوریتم پاسخ برخورد برای تماسی با کمترین سرعت دور شدن.

۴ـ اگر تکرارهای بیشتری وجود دارد، الگوریتم به گام اول باز می‌گردد.

الگوریتم به صورت خودکار، تمام تماس‌هایی را که قبلا تحلیل شده اند، دوباره بررسی می‌کند و تماس هایی را که در حال جدا شدن هستند، نادیده می‌گیرد. در هر تکرار، سخت‌ترین برخورد تحلیل می‌شود.

تعداد دفعات تکرار، باید حداقل برابر تعداد تماس‌ها باشد (تا هر کدام حداقل یک بار فرصت بررسی شدن را داشته باشند.). در شبیه‌سازی‌های ساده، تعداد تکراری برابر تعداد تماس‌ها به خوبی عمل می‌کند. در این جا، تعداد تکرارها به طور تخمینی، دو برابر تعداد تماس‌ها در نظر گرفته شده است. اما در مورد تماس‌های پیچیده، یقینا به تعداد بیشتری تکرار نیاز است. همچنین می‌توان روی الگوریتم، هیچ محدودیتی قرار نداد تا بتوان نحوه عملکرد آن را بررسی کرد.

تا این جا، از تداخل چشم پوشی کردیم. می‌توان تحلیل تداخل را با تحلیل برخورد ترکیب کرد. البته راه حل بهتر، جداسازی این دو تحلیل در دو فاز مجزا است. ابتدا، برخوردها را با استفاده از الگوریتم قبلی، به ترتیب تحلیل می‌کنیم. سپس، تداخل‌ها را تحلیل می‌کنیم.

جداسازی دو گام تحلیل، این امکان را فراهم می‌آورد که از ترتیبی متفاوت برای تحلیل تداخل استفاده کرد و بر اساس سرعت، تماس‌ها را مرتب نکرد. در این جا نیز، نیاز به نتایج واقع گرایانه داریم. می‌توان تماس‌ها را براساس شدت و سختی مرتب کرد و سپس تحلیل را انجام داد. اگر این دو مرحله را با هم ادغام کنیم، ترتیب برای یکی از دو تحلیل، ترتیب مطلوبی نخواهد بود.

تحلیل تداخل، ازالگوریتمی مشابه الگوریتم تحلیل برخورد استفاده می‌کند. همانند قبل، در هر تکرار، نیاز به محاسبه دوباره تمامی‌مقادیر نفوذ داریم. این نکته حائز اهمیت است که مقادیر تداخل توسط یابنده برخورد فراهم می‌شود. از آن جایی که اجرای الگوریتم تشخیص برخورد زمان بر است، نمی‌توان در پایان هر

تکرار آن را اجرا کرد. در نتیجه به منظور به روز رسانی مقادیر تداخل، مقدار جابه‌جایی دو جسم در تکرار قبلی را ذخیره می‌کنیم. سپس دو جسم در هر تماس بررسی می‌شوند، اگر یکی از دو جسم و یا هر دو، در فریم قبلی جابه جا شده باشد، مقدار تداخل با یافتن مولفه حرکت در جهت نرمال تماس، به روز می‌شود.

با در نظر گرفتن تمام این موارد، تابع تحلیلگر تماس به صورت زیر خواهد بود:

```
function ParticleContactResolver(iterations)
{
   this.iterations = iterations;
   this.iterationsUsed = 0;
}
ParticleContactResolver.prototype.resolveContacts =
function(contactArray,numContacts,duration){
   this.iterationsUsed = 0;
   while(this.iterationsUsed < this.iterations)
   {
      var max = 0;
      var maxIndex = numContacts;
      for (var i = 0; i < numContacts; i++)
      {
         var sepVel = contactArray[i].calculateSeparatingVelocity();
         if (sepVel < max)
         {
            max = sepVel;
            maxIndex = i;
         }
      }
      contactArray[maxIndex].resolve(duration);
      this.iterationsUsed++;
   }
}
```

ممکن است تعداد دفعات تکرار به کار برده شده در تحلیل تداخل با تعداد دفعات تکرار استفاده شده در تحلیل برخوردها یکسان نباشد. می‌توان پیاده‌سازی تابع را به گونه‌ای انجام داد که هر تحلیل، تعداد دفعات تکرار خاص خود را داشته باشد.

در عمل، نیازی به استفاده از دو مقدار متفاوت نیست. هر چه شبیه‌سازی دشوارتر شود و اجسامی‌در حال تعامل وجود داشته باشد، تعداد دفعات تکرار تحلیل برخورد، تقریبا به میزان افزایش تعداد دفعات تحلیل تداخل، افزایش می‌یابد. در پیاده‌سازی انجام شده در بالا، برای هر دو تحلیل، تعداد تکرار یکسان در نظر گرفته شده است.

محاسبه مجدد سرعت نزدیک شدن و مقدار تداخل، زمان برترین قسمت این الگوریتم است. در صورت وجود تعداد تماس زیاد، سرعت اجرای موتور فیزیک، تحت شعاع قرار می‌گیرد. در عمل، از آن جایی که ممکن است یک تماس بر تماس‌های دیگر اثری نداشته باشد، بسیاری از به روز رسانی‌ها بی فایده خواهند بود.

رویکرد دیگری در ایجاد یک موتور فیزیک قابل استفاده است. این رویکرد از لزوم تحلیل تداخل با ایجاد ترتیبی برای تحلیل تماس‌ها، اجتناب می‌کند. می‌توان به جای به روز رسانی موتور فیزیک در هر فریم، به روز رسانی‌های متعددی بر مبنای برخوردها انجام داد:

این شیوه به صورت زیر است:

در صورت عدم وجود برخورد، اجسام بر اساس قوانین حرکت و مولدهای نیرو، آزادانه به اطراف حرکت می‌کنند.

زمانی که برخوردی روی می‌دهد. نقطه برخورد، دقیقا در نقطه‌ای است که دو جسم با هم تماس پیدا می‌کنند. در این مرحله، هیچ تداخلی وجود ندارد.

اگر بتوان زمان دقیق برخورد را پیش بینی کرد، ابتدا از قوانین معمول حرکت استفاده می‌کنیم، سپس محاسبات مربوط به ضربه را انجام می‌دهیم و دوباره قوانین مربوط به حرکت نرمال را آغاز می‌کنیم.

اگر برخوردهای متعددی رخ دهد، این برخوردها را به ترتیب پردازش می‌کنیم. در زمان بین دو برخورد، با استفاده از قوانین نرمال حرکت، تمام اجسام را به روز رسانی می‌کنیم.

در واقع، این نوع موتور، از الگوریتم زیر پیروی می‌کند:

۱ـ زمان شروع را برابر زمان کنونی شبیه‌سازی و زمان پایان را برابر پایان زمان تقاضای به روز رسانی فعلی قرار می‌دهیم.

۲ـ به روز رسانی کاملی برای کل فاصله زمانی انجام می‌دهیم.

۳ـ یابنده برخورد را اجرا می‌کنیم و لیست برخوردها را جمع آوری می‌کنیم.

۴ـ اگر هیچ برخوردی صورت نگرفته باشد، از الگوریتم خارج می‌شویم.

۵ـ برای هر برخورد، زمان دقیق اولین برخورد محاسبه می‌شود.

۶ـ اولین برخوردی که باید صورت گیرد، انتخاب می‌شود.

۷ـ اگر اولین برخورد پس از زمان پایان رخ دهد، از الگوریتم خارج می‌شویم.

۸- به روز رسانی کامل مرحله دو، حذف می‌شود و به روز رسانی دیگری از زمان شروع تا زمان اولین برخورد انجام می‌شود.

۹ـ برخورد با اعمال ضربه‌های مناسب پردازش می‌شود (به تحلیل تداخل نیازی نیست چرا که در زمان برخورد دو جسم فقط در تماس با هم هستند.).

۱۰ـ زمان شروع را برابر زمان اولین برخورد قرار می‌دهیم، زمان پایانی را تغییر نمی‌دهیم و به گام اول باز می‌گردیم.

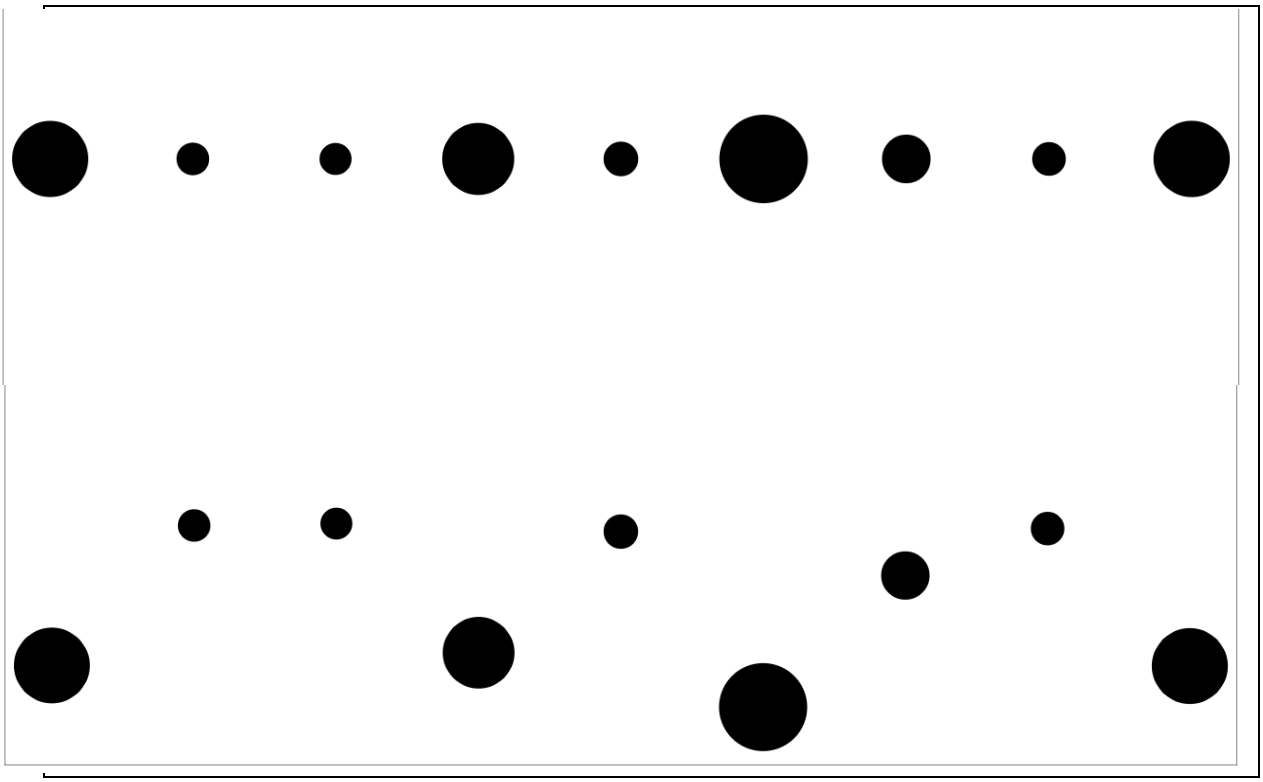
این الگوریتم، نتیجه دقیقی باز می‌گرداند و مشکلات مربوط به تحلیل تداخل را ندارد. این الگوریتم معمولا در برنامه‌های مهندسی فیزیک که دقت از اهمیت بالایی برخوردار است، به کار می‌رود. اما متاسفانه، این الگوریتم بسیار زمان بر است.

برای هر برخورد، یابنده تشخیص، اجرا می‌شود. همچنین، هر بار به روز رسانی فیزیک دوباره اجرا می شود. اما هنوز هم نیاز به کدی داریم که مدیریت تماس‌هاس ایستا را انجام دهد. در غیر این صورت، در هر تکرار، تماس ایستا به عنوان اولین برخورد، بازگردانده می‌شود. حتی اگر تماس ایستا موجود نباشد، خطاهای عددی در محاسبات مربوط به تشخیص برخورد، می‌تواند دوره‌های بدون پایان ایجاد کند.

در پروژه‌های مربوط به بازی، این روش عملی نیست. به روز رسانی در هر فریم، راه حل بهتری است، زیرا تمام تماس‌ها از نظر تداخل و سرعت تحلیل می‌شوند. تنها بازی‌هایی که امکان استفاده از این روش دارند، بازی‌های بیلیارد و اسنوکر هستند. در این بازی‌ها ترتیب برخوردها و موقعیت توپ‌ها در زمان برخورد بسیار مهم است. در شبیه‌سازی‌های با اهمیت، باید از الگوریتم قبلی استفاده کرد.

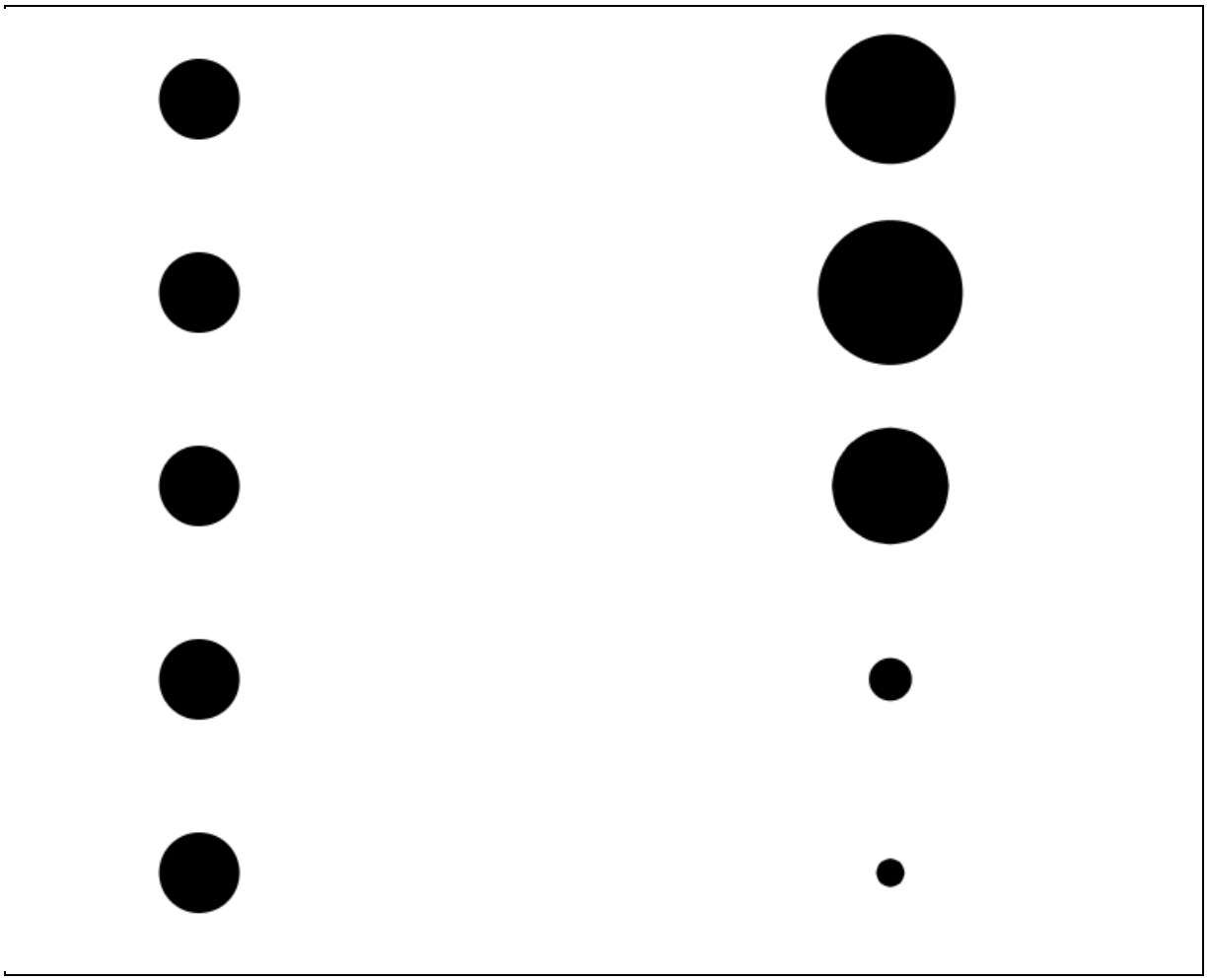
در تصاویر زیر مثال‌هایی از اعمال محدودیت‌های قوی را می بینیم.

در مثال زیر اثر جنس جسم و در واقع ضریب ارتجاع بر سرعت و رفتار جسم با ضریب‌های متفاوت را می بینیم. همان طور که مشاهده می شود جسم با ضریب بالاتر (در این جا با اندازه کوچتر) ارتجاع بالاتری دارد:

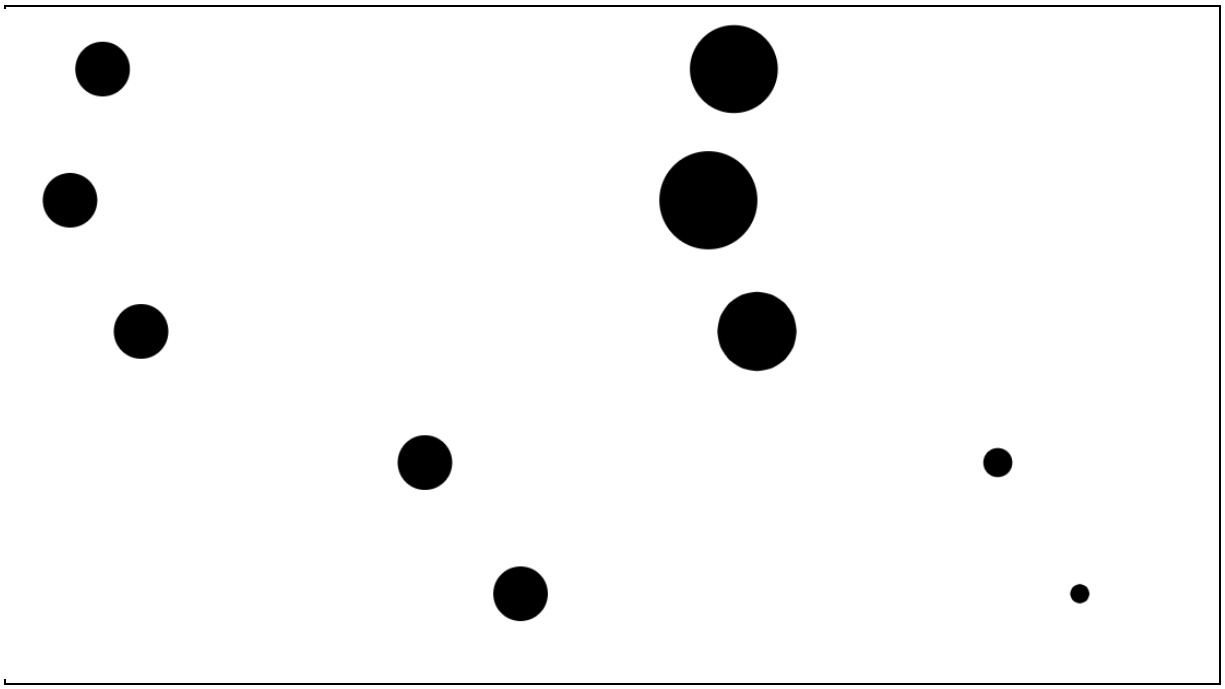


شکل ۱۸ تحلیل برخورد با ثابت متفاوت

در یک مثال دیگر اثر جرم جسم بر سرعت بعد از برخورد جسم مشاهده می شود:



شکل ۱۹برخورد با جرم‌هایی متفاوت



شکل ۲۰ بعد از برخورد

۳.۸ پدیده‌های برخوردگونه

همان طور که در مورد فنر دیدیم، به چند نوع از اتصالات که با استفاده از تکنیک‌های مطرح شده، قابل مدل کردن هستند، نگاهی می‌اندازیم.

می‌توان برخورد را تلاشی برای جدا نگه داشتن دو جسم در فاصله‌ای بسیار کم، تصور کرد. تماس زمانی بین دوجسم ایجاد می‌شود که دو جسم به شدت به هم نزدیک شوند. می‌توان از تماس برای نگه داشتن دو جسم در کنار هم استفاده کرد.

کابل محدودیتی است که دو جسم را مجبور می‌کند به اندازه طول کابل از هم فاصله داشته باشند. اگر از کابلی ضعیف استفاده شود، دو جسم تا زمانی که به هم نزدیک هستند، اثر کابل را حس نمی‌کنند. اما اگر کابل محکم کشیده شود، دو جسم بیشتر از طول کابل از هم دور نخواهند شد. بسته به خصوصیات و ویژگی‌های

کابل، اجسام ممکن است کمی‌بیش‌تر یا کم‌تر از این طول، به همان شیوه‌ی اجسام در حال برخورد، جابه جا شوند. کابل دارای یک ضریب ارتجاع است که مقدار جابه‌جایی را کنترل می‌کند.

کابل را می‌توان با ایجاد تماس در زمانی که دو انتهای آن به مقدار زیاد کشیده می‌شود، مدل کرد. این تماس بسیار شبیه تماس‌های استفاده شده در رابطه با برخوردها است، با این تفاوت که نرمال تماس عکس می شود. نرمال تماس در این حالت، به اجسام در جهت نزدیک شدن فشار می‌آورد، نه در جهت دور شدن. مقدار تداخل تماس هم به مقداری که کابل بیش‌تر از حد خود کشیده شود، مرتبط است.

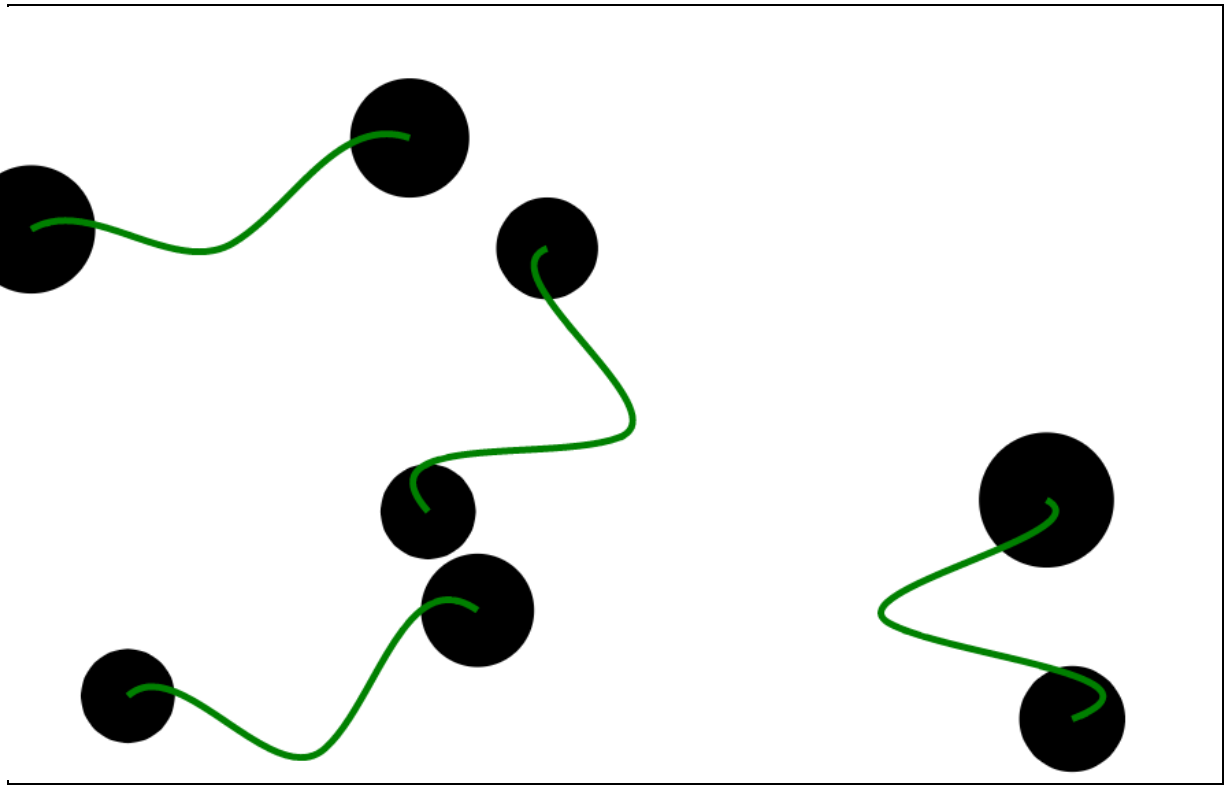
پیاده‌سازی مولد تماس برای کابل به صورت زیر است:

```
function ParticleLink()
{
   this.particle = new Array();
}
ParticleLink.prototype.currentLength = function(){
   var relativePos = new Point(this.particle[0].position.x -
this.particle[1].position.x,this.particle[0].position.y - this.particle[1].position.y);
   return relativePos.length;
}
ParticleContact.prototype.fillContact = function(contact,limit){
   return 0;
}
function ParticleCable()
{
   ParticleLink.call(this);
   this.maxLength = null;
   this.restitution = null;
}
ParticleCable.prototype = new ParticleLink();
ParticleCable.prototype.constructor = ParticleCable;
ParticleCable.prototype.fillContact = function(contact, limit){
   var length = this.currentLength();
   if (length < this.maxLength)
   {
      return 0;
   }
   contact.particle[0] = this.particle[0];
```

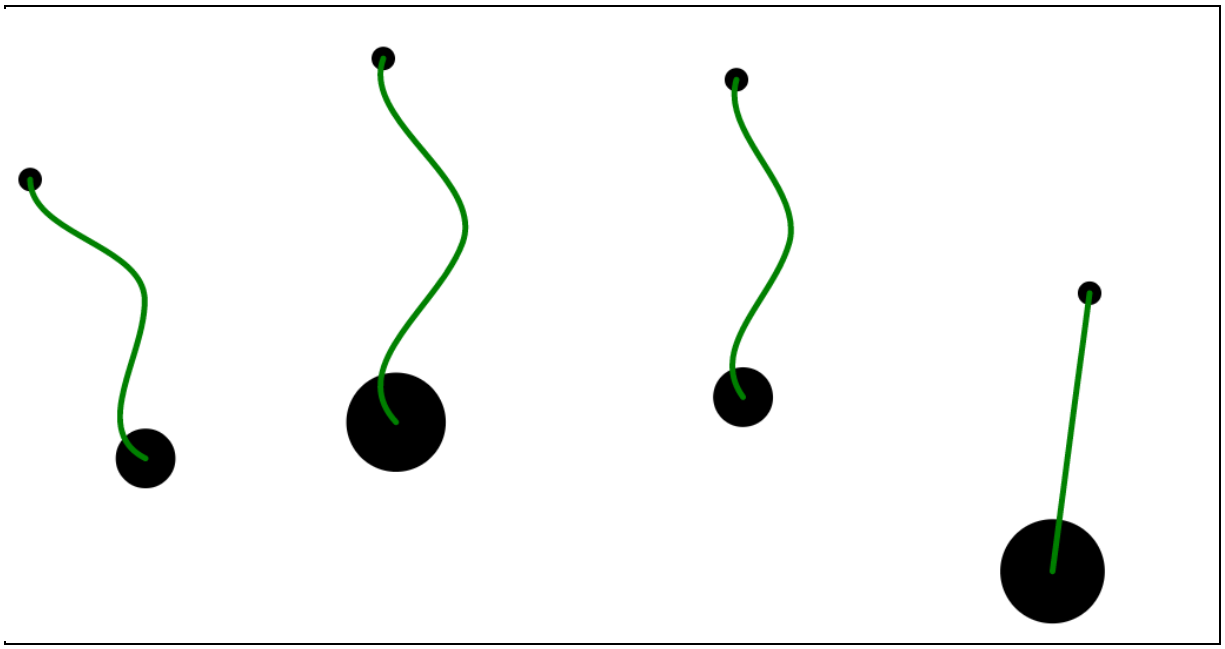
```
    contact.particle[1] = this.particle[1];
    var normal = new Point(this.particle[1].position.x -
this.particle[0].position.x,this.particle[1].position.y - this.particle[0].position.y);
    normal = normal.normalize();
    contact.contactNormal = normal;
    contact.penetration = length-this.maxLength;
    contact.restitution = this.restitution;
    return 1;
}
```

کد بالا به عنوان یک یابنده برخورد عمل می‌کند: حالت فعلی کابل را بررسی می‌کند و اگر کابل به حد انتهایی خود رسیده باشد، یک تماس برمی‌گرداند. این تماس به تماس‌های دیگری که توسط یابنده برخورد ایجاد شده است، اضافه می‌شود و توسط الگوریتم تحلیلگر نرمال تماس پردازش می‌شود.

در تصاویر زیر مثالی از پیاده‌سازی چند کابل و ذرات متصل به آن‌هار امی توانید مشاهده کنید:



شکل ۲۱مثال اول طناب‌ها



شکل ۲۲ مثال دوم طناب‌ها

میله رفتار کابل و برخورد را ترکیب می‌کند. دو جسمی‌که توسط میله به هم وصل شده اند، نه می‌توانند به هم نزدیک شوند و نه از هم دور شوند. دو جسم، در یک فاصله‌ی مشخصی از هم قرار دارند. پیاده‌سازی میله مشابه پیاده‌سازی مولد تماس کابل خواهد بود. در هر فریم، حالت فعلی میله بررسی می‌شود و تماسی ایجاد می شود که دو انتها را یا به هم نزدیک می‌کند و یا دو انتها را ازهم دور می‌کند.

در این جا نیاز به تغییر دو مورد که قبلا به آن اشاره شد، است. اول این که، ضریب ارتجاع همیشه برابر صفر خواهد بود. زیرا دو انتها در هیچ جهتی حرکت و لرزش ندارند. از آن جایی که این دو در فاصله‌ی ثابتی از هم قرار می‌گیرند، سرعت نسبی بین آن‌ها برابر صفر است. دوم این که، اگر در هر فریم، تنها یکی از دو تماس (دور شدن یا نزدیک شدن) را اعمال کنیم، میله به لرزش در خواهد آمد. در فریم‌های متوالی، این امکان وجود دارد که ابتدا میله کوتاه و سپس بلند شود و هر تماس میله را به عقب یا جلو بکشاند. برای جلوگیری از وقوع این حالت، در هر فریم، هر دو تماس را ایجاد می‌کنیم. اگر به یکی از تماس‌ها نیاز نباشد (مثلا، سرعت دور شدن

بزرگتر از صفر باشد یا تداخل وجود نداشته باشد.)، از آن چشم‌پوشی می‌شود. وجود تماس اضافی، به الگوریتم تحلیلگر تماس کمک می‌کند تا میله پایدارتر شود.

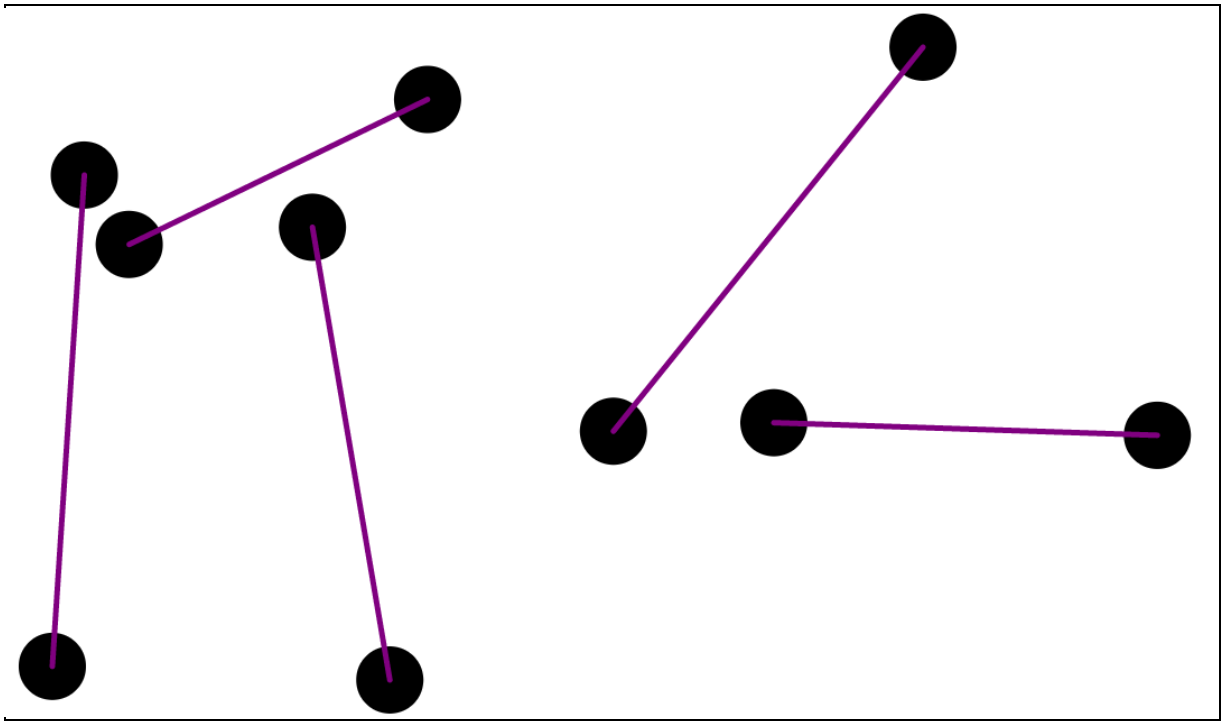
این روش در زمانی که مجموعه‌ای پیچیده از میله‌ها وجود داشته باشد، بسیار ضعیف عمل می‌کند، چراکه تعداد دفعات تکرار مورد نیاز، برای رسیدن به حالتی پایدار به شدت افزایش می‌یابد. اگر تعداد دفعات تکرار کم باشد، لرزش دوباره مشاهده خواهد شد.

پیاده‌سازی مولد تماس به صورت زیر خواهد بود:

```
ParticleRod.prototype.fillContact = function(contact,limit){
   var currentLen = this.currentLength();
   if (currentLen == this.length)
   {
      return 0;
   }
   contact.particle[0] = this.particle[0];
   contact.particle[1] = this.particle[1];
   var normal = new Point(this.particle[1].position.x -
this.particle[0].position.x,this.particle[1].position.y - this.particle[0].position.y);
   normal = normal.normalize();
   if (currentLen > this.length) {
      contact.contactNormal = normal;
      contact.penetration = currentLen - this.length;
   } else {
      contact.contactNormal = new Point(normal.x * -1,normal.y *-1);
      contact.penetration = this.length - currentLen;
   }
   contact.restitution = 0;
   return 1;
}
```

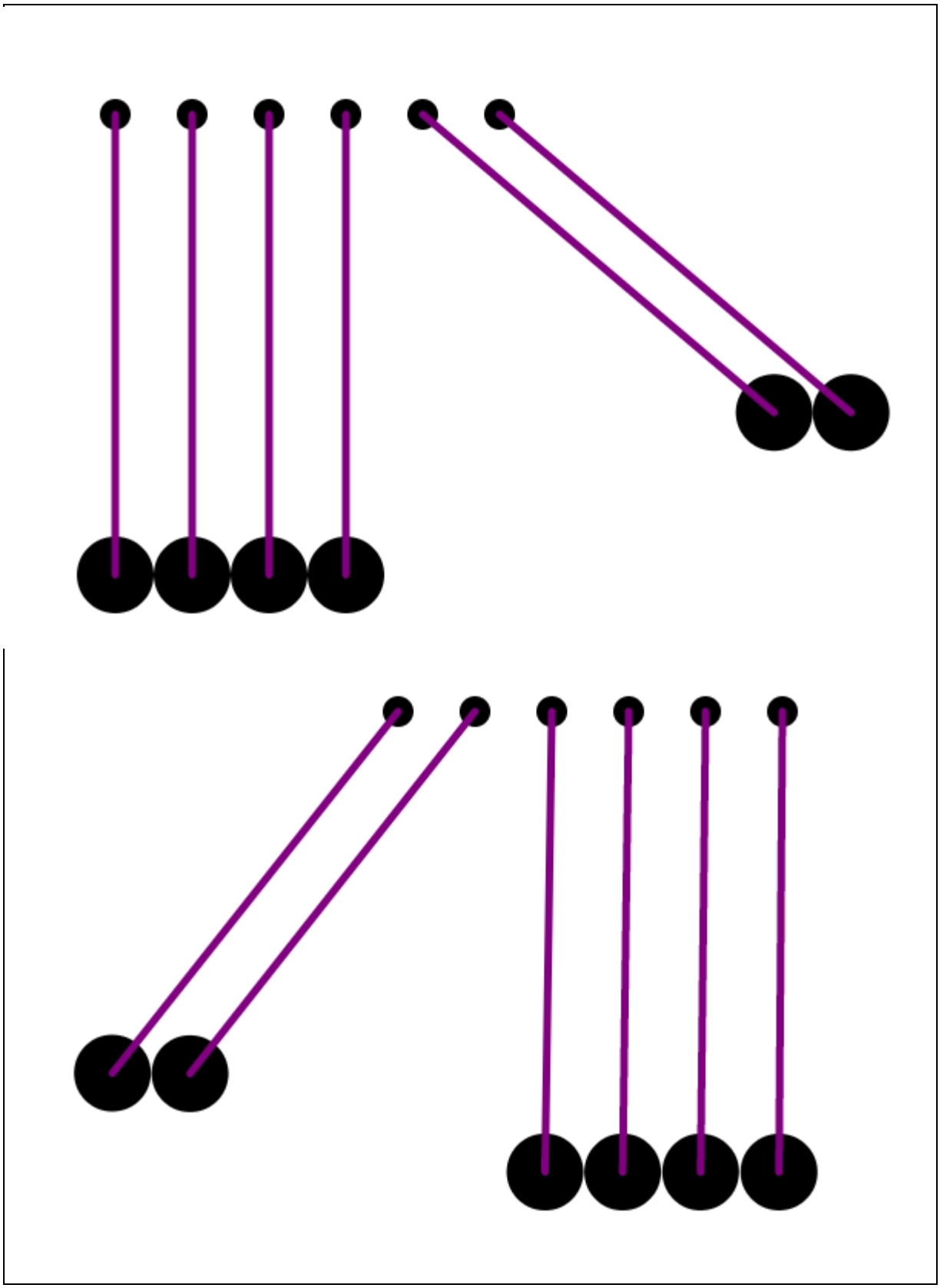
کد بالا، همواره دو تماس تولید می‌کند که باید به لیست تماس‌های برگردانده شده توسط یابنده برخورد اضافه شود و به تحلیلگر تماس تحویل داده شود.

در تصویر زیر نمایشی از یک مثال میله‌ها آمده است:



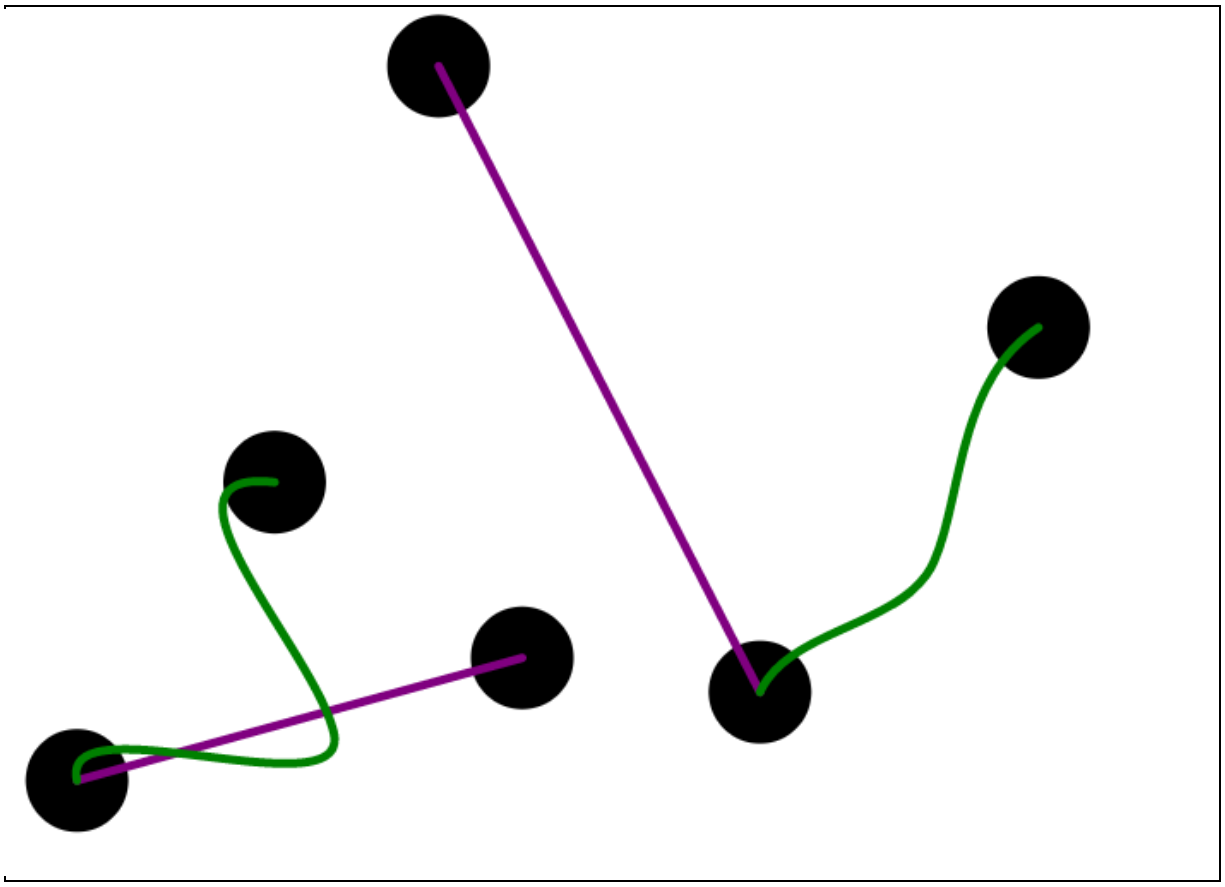
شکل ۲۳ مثال میله‌ها

همچنین یک مثال کلاسیک از حرکت آونگ با کمک میله‌ها پیاده‌سازی شده که تصویر آن را در زیر مشاهده می کنید:



شکل ۲۴ مثال آونگ

یک مثال ترکیبی از میله‌ها و کابل‌ها را هم تصویر زیر آورده شده است:



شکل ۲۵ مثال ترکیبی

۴.۸ جمع‌بندی و نتیجه‌گیری

در این فصل آخرین قسمت از موتور فیزیک یعنی محدودیت‌های قوی را بررسی کردیم و نشان دایدم که چطور برخوردها و پدیده‌های شبیه برخورد نقش حیاتی در ایجاد موتور تجمیع اجرام داشته باشند. انواع تولید برخوردها و تحلیل آن‌ها حالا در موتور انجام می شود و همانطور که در فصل بعد خواهیم دید به راحتی می توان از آن‌ها استفاده کرد تا یک موتور تجمیع کامل را در اختیار گرفت.

فصل نهم: استفاده از موتور فیزیک تجمیع جرم

۱.۹ مقدمه

تا این جا، موتور فیزیک تجمیع اجرامی‌ساخته شد که توانایی شبیه‌سازی ذره و ایجاد اجسامی‌از اشیا متعددی که توسط میله، کابل و فنر به هم متصل شده اند،تشکیل شده اند را دارد. حال زمان آن رسیده که این موتور را بر روی چند سناریو آزمایش کنیم. موتور پیاده‌سازی شده، هنوز محدودیت‌هایی دارد، برای نمونه، این موتور توانایی توصیف نحوه چرخش اجسام را ندارد. با استفاده از تجمیع اجرام، چرخش اجسام را به شکل ساختگی ایجاد می‌کنیم. این تکنیک، در مورد برخی از برنامه‌ها بسیار کارآمد است و دیگر نیازی به استفاده از موتورهای پیشرفته، نخواهد بود.

۲.۹ نمای کلی موتور

موتور فیزیک تجمیع اجرام دارای سه جزء است:

۱ـ خود ذرات، موقعیت، حرکت و جرم خود را ثبت می‌کنند. برای ایجاد یک شبیه‌سازی، باید بدانیم چه ذراتی نیاز است و سرعت مکان اولیه آن‌ها را نیز مشخص کنیم. همچنین جرم معکوس این ذرات نیز باید مقدار دهی شود. شتاب جسم که ناشی از جاذبه زمین است، در جسم صلب نگه داری می‌شود (می‌توان آن را حذف کرد و با نیرویی جایگزین کرد.).

۲ـ مولدهای نیرو برای نگه داری و ثبت نیروهایی که در طول فریم‌های متوالی اعمال می‌شوند، به کار می‌رود.

۳ـ سیستم برخورد، مجموعه‌ی تماس‌های جسم را جمع آوری می‌کند و آن را به تحلیلگر برخورد تحویل می‌دهد. در این موتور فیزیک، دو مولد تماس در نظر گرفته شده است: یابنده برخورد و فشار کابل یا میله.

در هر فریم، اطلاعات داخلی تمام ذرات را محاسبه می‌کنیم، مولدهای نیروی ذرات را فراخوانی می کنیم. سپس integrator به منظور به روز کردن موقعیت و سرعت ذرات فراخوانی می‌شود. در انتها، تمام تماس‌های ذره جمع آوری می‌شود و به تحلیلگر برخورد تحویل داده می‌شود.

به منظور آسان‌تر کردن این فرایند، ساختاری ساده ایجاد می‌کنیم که توانایی نگه داری هر تعداد جسم صلب را داشته باشد. این اجسام صلب در یک linked list به همان شیوه به کار رفته در مورد مولدهای نیرو، نگه داری می‌شوند. این linked list در یک کلاس World قرار دارد که نمایانگر تمام دنیای شبیه‌سازی شده فیزیکی است:

```
function ParticleRegistration()
{
   this.particle = null;
   this.next = null;
}
function ParticleWorld(maxContacts,iterations)
{
   this.firstParticle = null;
}
At each frame the startFrame method is first called, which sets up each object
ready for the force accumulation:
ParticleWorld.prototype.startFrame = function(){
   var reg = this.firstParticle;
   while (reg)
   {
      reg.particle.forceAccum.x = 0;
      reg.particle.forceAccum.y = 0;
      reg = reg.next;
   }
}
```

باقی نیروها می توانند بعد از فراخوانی این تابع اعمال شوند.

همچنین سیستم دیگری برای نگهداری برخوردها داریم. همان طور که برای مولدهای نیرو هم همین کار را کردیم از رابط‌های پلی مورفیک برای یابنده‌های برخورد استفاده می‌کنیم.

```
function ParticleContactGenerator()
{
}
ParticleContactGenerator.prototype.addContact = function(contact,limit){

}
```

هر یک از این توابع به نوبه خود از درون World فراخوانی می‌شوند و هر تماسی را که یافته می‌شود، با فراخوانی addContact به World تحویل داده می‌شود.

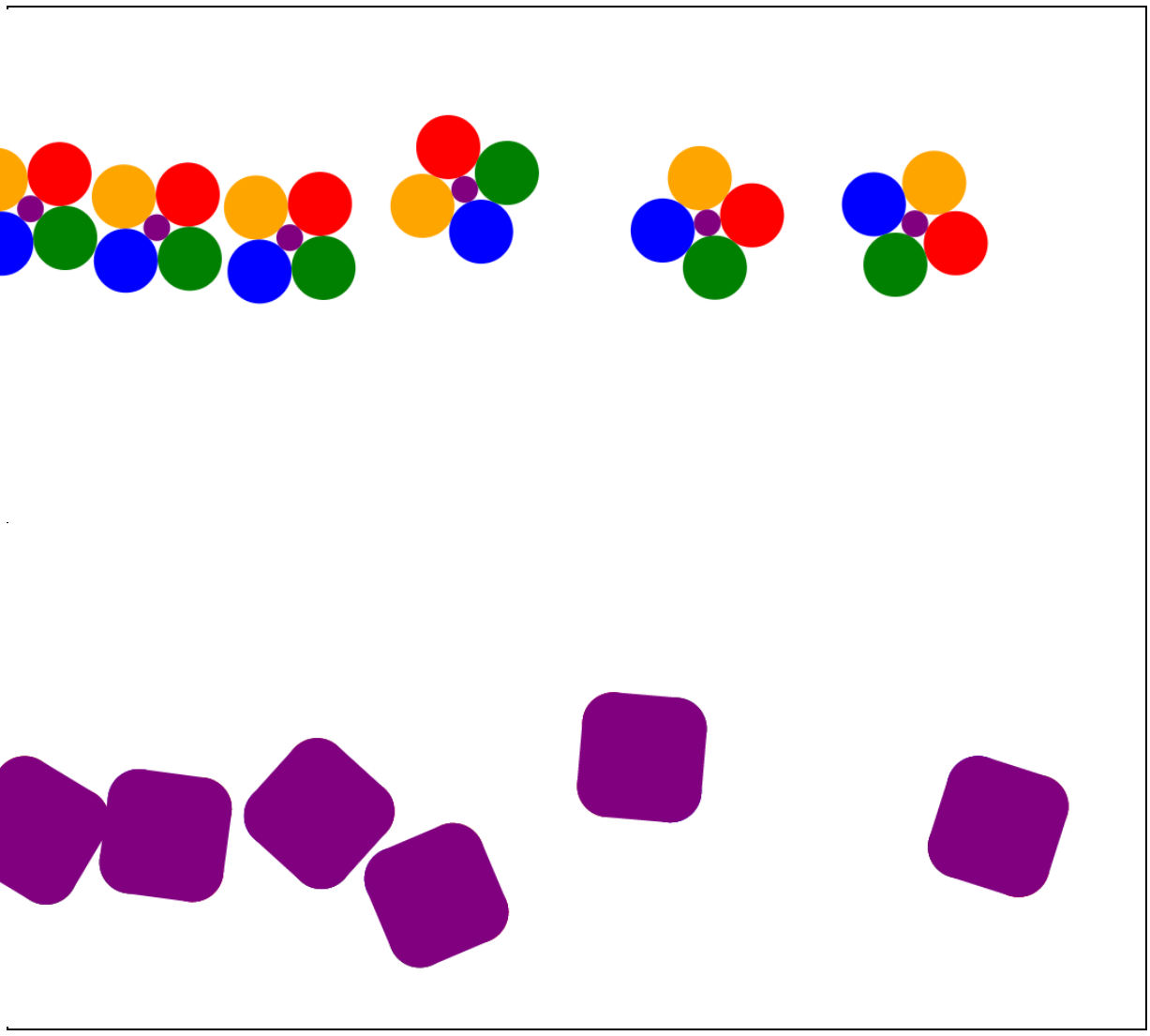
برای اجرای این موتور فیزیک، تابع runPhysics فراخوانی می‌شود. این تابع، تمام مولدهای نیرو را با هدف اعمال نیرو فراخوانی می‌کند و سپس اجتماع تمام اجسام و یابنده تماس را اجرا می‌کند و لیست نتیجه تماس‌ها را تحلیل می‌کند.

۳.۹ جمع‌بندی و نتیجه‌گیری

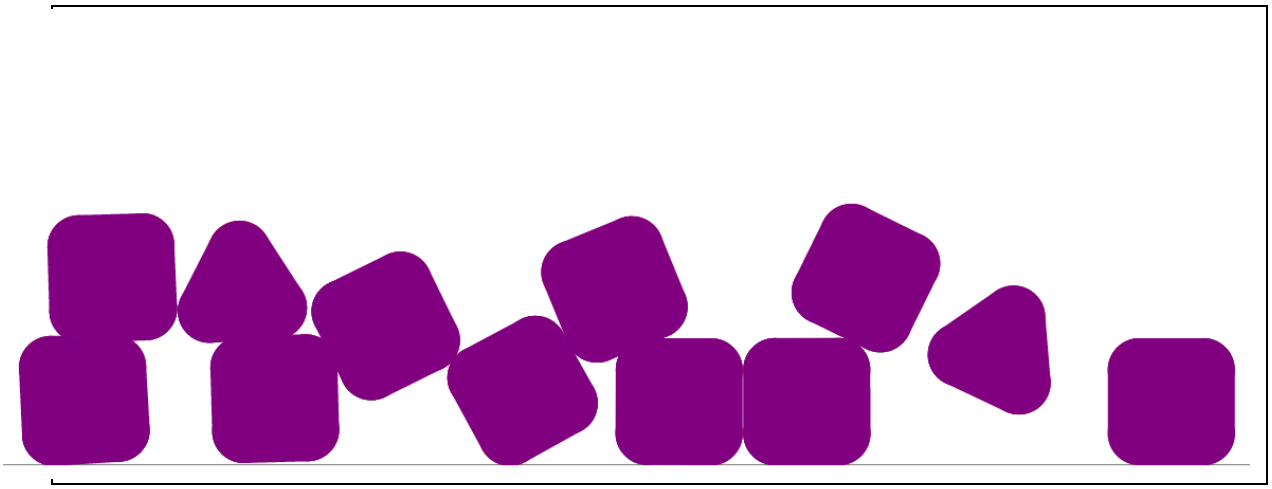
برنامه‌های کاربردی زیادی در رابطه با موتور تجمیع اجرام می‌توانند ایجاد شوند: ایجاد ساختارهایی از اجرام ذره و محدودیت‌های قوی. با استفاده از این تکنیک، می‌توان اجسام بزرگ‌تر بسیاری را ایجاد و شبیه‌سازی کرد. برای نمونه، جعبه‌ها، دستگاه‌های مکانیکی، و حتی زنجیرها و ماشین آلات. اگر فنرها را هم در نظر بگیریم، حباب‌های نرم و دگر دیس پذیر را هم می‌توان اضافه کرد.

در نتیجه یک موتور فیزیک تجمیع اجرام، توانایی شبیه‌سازی چند اثر جالب و پیچیده را دارد. به ویژه، مجموعه‌ای از اجسام نسبتا ساده که توسط ترکیبی از محدودیت‌های قابل ارتجاع و سخت، به هم متصل شده اند، نمونه خوبی از استفاده از این رویکرد هستند.

در تصاویر زیر مثال‌هایی مهم از قابلیت‌های این موتور در ایجاد و مدیریت اجسامی که از تجمیع ذرات ایجاد شده اند آورده شده است توجه شود که برای اتصال ذرات از میله‌ها استفاده شده است تا بتوان از چند ذره یک جسم شبیه صلب داشت که حتی بتواند علارقم عدم وجود مکانیزیم درونی چرخش در موتور بچرخد. دو نوع شکل مربع و مثلث تشکیل شده از دایره را در تصاویر زیر مشاهده می کنید:

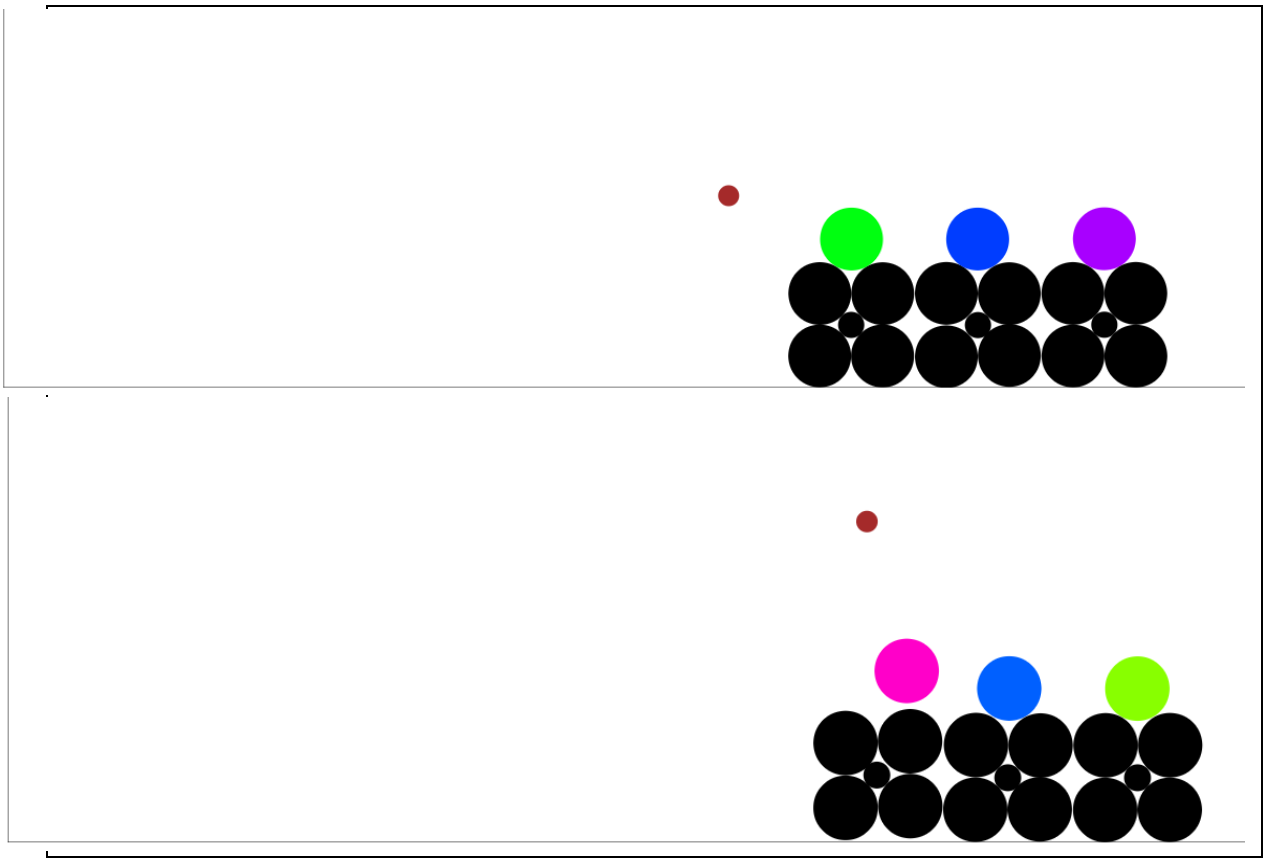


شکل ۲۶ مثال تجمیع اجرام



شکل ۲۷ انواع اجرام تجمیعی

در تصویر زیر همچنان یک بازی ساده ایجاد شده توسط این موتور که در آن بازیکن با پرتاب توپی اهدافی را می زند مشاهده می کنید:



شکل ۲۸ یک بازی با موتور

نتیجه‌گیری و پیشنهادها

HTML5 به عنوان جدیدترین نسخه زبان ابرمتنی استاندارد وب در راستای تحقق بیشتر اهداف اولیه اش دارای عناصر بسیاری جدید مهمی‌است که انتظار می‌رود با فراگیر شدن آن بسیاری از مسائل و مشکلات موجود در نمایش و انتقال اطلاعات در وب که تا کنون فضای اینترنت با آن‌ها روبرو بوده است را به سمت راه حل‌هایی استاندارد و همه گیر ببرد. یکی از مهمترین این عناصر عنصر canvas می‌باشد که در واقع فضایی را در اسناد HTML ایجاد می‌کند که بتوان در آن انواع رسوم گرافیکی برداری را داشت. آن چه این عنصر را قدرتمند می‌کند قابلیت آن برای پذیرش زمینه‌های مختلف است. دو زمینه اصلی موجود برای canvas، 2D Context و webgl می‌باشند که هر کدام رابط کاربری خود را برای کاربر ارائه می‌دهند. این رابط‌های کاربری به کاربر اجازه می‌دهند که از محیط canvas استفاده کنند و در آن به رسم تصاویر برداری بپردازند. در این پروژه تمرکز اصلی بر روی زمینه 2D بوده و به توابع و امکانات آن توجه شده است.

مهمترین نکته در کار با زمینه‌ها ابراز رابط کاربری آن‌ها در زبان JavaScript است، این رابط کاربری با کمک آنچه مرورگر با اسکریپت نویسی به برنامه نویس اجازه می‌دهد، این امکان را به وجود می‌آورد تا بتوان در محیط Canvas انیمیشن و بیشتر از آن بازی‌های ویدئویی ساخت و اجرا کرد. این موضوع سبب شد تا بسیاری از برنامه نویسان تحت وب روی به استفاده از این عنصر و امکانات برای ساخت و ایجاد بازی‌های ویدئویی بیاورند. اما به علت نو بودن تمام این تکنولوژی می‌توان گفت که هنوز ابزارهای مناسب و با کیفیتی که این پروسه را سرعت ببخشند ایجاد نشده‌اند. یکی از مهمترین این ابزارها در طراحی و ایجاد بازی‌های ویدئویی موتور فیزیک می‌باشد.

موتور فیزیک در واقع قطعه کدی است که وظیفه شبیه‌سازی پدیده‌های فیزیکی در عالم واقع را در بازی‌های ویدئویی دارد. بازی‌های ویدئویی از این موتور استفاده می‌کنند تا آن چه نشان می‌دهند برای بازیکن طبیعی‌تر و جذاب‌تر باشد. بازی‌هایی که نیاز دارند تا آنچه در دنیا رخ می‌دهد را به بازیکن نشان دهند باید از

نحوه کار فیزیک اطلاع داشته و ریاضیات آن را در کد خود پیاده‌سازی کرده باشند. این ریاضیات می‌توانند به طور کلی در کنار بازی قرار بگیرند و نحوه به روز رسانی هر یکی از اشیا در بازی را مشخص کنند. این ویژگی سبب می‌شود که بتوان این قطعه کد را از بازی جدا کرد و به طور جداگانه به گسترش و بالا بردن کیفیت آن پرداخت، در نتیجه نه یک بازی که هر بازی که بخواهد از این قابلیت‌ها استفاده کند می‌تواند از یک موتور جداگانه استفاده کند.

در این پروژه یک موتور فیزیک با قابلیت‌های متنوعی برای استفاده در بازی‌هایی که در محیط canvas ساخته می‌شوند، پیاده‌سازی کردیم. برای پیاده‌سازی این موتور از فیزیک ذرات شروع کرده و با شبیه‌سازی حرکت آن‌ها یک موتور فیزیک ذرات ساختیم. سپس با اضافه کردن مولدهای نیرو و تاثیر تجمیعی آن‌ها انواع پدیده‌های نیرو زا و وارد کننده نیرو را به موتور اضافه کرده و تاثیر آن‌ها را بر ذرات پیاده‌سازی کردیم. آن چه که در مورد نیرو و به خصوص نیروهای فنری بحث کردیم روشن کرد که این نیروها در تعداد بسیار زیادی از پدیده های طبیعی وجود دارند و در شبیه‌سازی آن‌ها می‌توانند به کار بیایند. اما آنچه بعدا با بررسی قدرت پردازشی کامپیوترها روشن شد این بود که در شبیه‌سازی محدودیت‌های قوی با آنکه این محدودیت‌ها بسیار شبیه فنرها عمل می‌کنند، نمی‌توان از این خاصیت استفاده کرد چرا که به خطاهایی بزرگ منجر می‌شود در نتیجه یک سیستم مدیریت برخورد به وجود آوردیم که مستقل از مول نیروها عمل می‌کند و برخوردها را شناسایی و سپس تحلیل می‌کند. این سیستم به ما کمک کرد تا بتوانیم انواع برخورها و پدیده‌های مثل برخورد مانند طناب‌ها و میله‌ها را شبیه‌سازی کنیم.

موتور فیزیکی که در نتیجه در این پروژه پیاده‌سازی شد یک موتور فیزیک تجمیع اجرام بوده که می تواند به عنوان یک موتور فیزیک کامل در بازی‌ها مورد استفاده قرار بگیرد اما این موتور همچنان می‌تواند پدیده های بیشتری در طبیعت را در بر بگیرد و حتی تبدیل به یک موتور اجسام صلب شود که در برخی بازی‌ها

قدرتمندتر عمل خواهد کرد. موتور اجسام صلب برای شبیه‌سازی اجسام بزرگ بسیار کاراتر خواهد بود و در آن به جای این که مانند این موتور اجسام را از ذرات بسازیم می‌توانیم آن‌ها را به طور یگانه و کلی در نظر بگیریم. برای این کار تغییراتی در موتور باید صورت بگیرد. مهمترین تغییرات نحوه چرخش اجسام است، این موضوع بابت خود باعث پیچیدگی‌های زیادی خواهد شد که باعث می‌شود از مجال این پروژه بیرون باشد اما در آینده می‌توان آن را پیاده‌سازی کرد و به این وسیله این موتور را قدرتمندتر ارائه داد.

منابع و مآخذ

[1] Y. Cao, "Phusis studio: A real-time physics engine for solid and fluid simulation," *Computational Problem-Solving (ICCP), 2011 International Conference on,* pp. 462 - 465, 2011.

[2] D. H. Eberly, Game Physics, Second Edition, Morgan Kaufmann, 2010.

[3] C. Ericson, Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology), Morgan Kaufmann, 2005.

[4] B. Frain, Responsive Web Design with HTML5 and CSS3, Packt Publishing, 2012.

[5] S. Fulton, HTML5 Canvas, O'Reilly Media, 2011.

[6] D. Geary, Core HTML5 Canvas: Graphics, Animation, and Game Development, Prentice Hall, 2012.

[7] R. Hawkes, Foundation HTML5 Canvas: For Games and Entertainment, friendsofED, 2011.

[8] G. Palmer, Physics for Game Programmers, Apress, 2005.

[9] M. Pilgrim, HTML5: Up and Running, O'Reilly Media, 2010.

[10] I. Millington, Game Physics Engine Development (Morgan Kaufmann Series in Interactive 3D Technology), Morgan Kaufmann, 2010.